# eProDas

## Data Acquisition system

## Users' guide and programming manual

Developed under the
Leonardo da Vinci Pilot Project

ComLab (part 2)

http://www.e-prolab.com/comlab/

**Written by:** **Boštjan Murovec**
**bostjan.murovec@fe.uni-lj.si**
**bostjan@lie.fe.uni-lj.si**

**Visual Basic API definitions by:** **Tomaž Kušar**
**tomaz.kusar@guest.arnes.si**

**Last changed on December 21th, 2007.**

**Please, read the Introduction carefully and comprehensively before starting to use this material.**

**Please, read the Known Issues at the end of the users' guide before reporting bugs and problems to us.**

# Developers' team

### Software and API development, some work on schematics:

Boštjan Murovec
University of Ljubljana
Faculty of electrical engineering
Tržaška cesta 25
1000 Ljubljana, Slovenia

Tel.: +386 1 4768 265
Fax: +386 1 4768 426
Email: bostjan.murovec@fe.uni-lj.si
bostjan@lie.fe.uni-lj.si
Web: http://lie.fe.uni-lj.si/staff/murovec_en.htm

### Schematics, co-ordination and concept designer:

Slavko Kocijančič
University of Ljubljana
Faculty of education
Kardeljeva ploščad 16
1000 Ljubljana, Slovenia

Tel.: +386 1 5892 296
Fax: +386 1 5892 233
Email: slavko.kocijancic@pef.uni-lj.si
Web: http://www.pef.uni-lj.si/slavkok/

### Schematics, testing and evaluation, LabView support

David Rihtaršič
University of Ljubljana
Faculty of education
Kardeljeva ploščad 16
1000 Ljubljana, Slovenia

Tel.: +386 1 5892 224
Fax: +386 1 5892 233
Email: david.rihtarsic@guest.arnes.si

### Schematics, testing and evaluation, Visual Basic support

Tomaž Kušar
University of Ljubljana
Faculty of education
Kardeljeva ploščad 16
1000 Ljubljana, Slovenia

Tel.: +386 1 5892 236
Email: tomaz.kusar@guest.arnes.si

## Acknowledgements

- Microchip Corporation is a registered company name (www.microchip.com).
- PIC is a registered trademark of Microchip Corporation.

- Microsoft Corporation is a registered company name (www.microsoft.com).
- Windows in all incarnations (including but not limited to 95, 98, NT, 2000, XP, 2003, Vista) is a registered trademark of Microsoft Corporation.
- Visual Basic is a registered trademark of Microsoft Corporation.

- Borland is a registered company name (www.borland.com).
- Delphi is a registered trademark of Borland Corporation.

- Linux is a registered name of Open Source Development Laboratories (www.osdl.org).

- Texas Instruments is a registered company name (www.ti.com).

- National Instruments is a registered company name (www.ni.com).
- LabView is a registered trademark of National Instruments Corporation.

- MathWorks is a registered company name (www.mathworks.com).
- Matlab is a registered trademark of MathWorks Corporation.

- Vernier is a registered company name (www.vernier.com).

# Table of contents

# List of Tables

# List of Figures

# Users' Guide

# 1 Introduction

eProDas aims to be an affordable but pertinent entry-level data acquisition system that is primarily suitable for performing a wide spectrum of natural science experiments as a part of pedagogical process. The related role of the system, which is intimately connected to the data acquisition, is to be an affordable alternative to the expensive instrumentation such as digital multimeters and to small extent even oscilloscopes, which are often out of reach for many potential users of these devices.

The system should be capable and flexible enough to be suitable for integration into all levels of education from primary schools to universities, which are our primary target group. Our second target group are pupils and students themselves as well as hobbyists and home made enthusiasts that need or want to have a fairly capable data acquisition system for an extremely low price. Namely, the parts of the simplest system within the eProDas set of devices cost around 20 €, which should be within the reach of everyone who wants to use the system and is willing to put some effort into building it. Still, parts' costs may not be the only expenses that arise from building the system. Therefore, please, read this users' guide in its entirety to get familiar with all the issues in order to reduce the chance of your disappointment at a later time.

The development of the eProDas system with the accompanying software and experiments is partially funded by Leonardo da Vinci Vocational Training Programme. All the outcomes are meant to be freely available to anyone and widely disseminated, primarily by means of Internet, where everyone can get all the related software and enough information to build the data acquisition hardware by him/her self. We kindly ask you to spread the information about eProDas to anyone interested.

Since all the outcomes are available to you free of charges, please note the following.

! **The developers of the system do not take any responsibilities whatsoever if the use or misuse of the system causes or leads to personal injuries, data loss, material damage or any other undesired consequences. Under no conditions should the developers of the system be charged or prosecuted by anyone using the system for any reason whatsoever. If you are not sure what you are doing, ask for help an experienced person or personnel.**

If you agree with the above paragraph and you are still willing to give the system a try, we kindly ask you to help as, too. If you find any bug, inconvenience, typing error in documentation or whatever you think should be corrected or improved, please let us know. We cannot promise you that every suggestion will be taken into account but features or corrections that are meaningful to a broad audience will be implemented or done within the limitations of our resources (primarily time).

Further, if you are a developer willing to dig into the source code and/or electrical schematics of the system and you expand the capabilities with new or improved features, which you would like to share with others, inform as about your work. If your propositions make sense for other users of the system, we will apply/include them to the current contents of the system with proper acknowledgement.

**The developers' team wishes you fruitful and pleasant use of the eProDas system. Enjoy.**

# 2 Overall description

The name eProDas denotes data-acquisition platform as well as the set of concrete devices that are capable of data acquisition and signal generation, both of which is needed for performing various experiments within the scope of natural sciences, such as physics, chemistry, biology, medicine, electrical engineering and so forth. Undoubtedly, there already exist many such systems, produced by respected firms with international repute on one side and by isolated small groups of developers or even hobbyists on the other side. The former are generally feature rich systems of a high quality but too expensive for massive use by usually under funded pedagogical institutions or do-it-yourself enthusiasts. The latter lack many features, they are not suitable for broad audience or they are not polished and mature enough to be useful for many potential users.

The eProDas aims to (at least partially) remedy the situation by combining the best features from both approaches. However, do not expect miracles, since we are not magicians and we cannot offer you a top-quality system for the negligible price. What we can do is offer you a fairly easy and cheap entrance to the world of data acquisition and measurements with the system that is suitable for many not-too-demanding practical cases.

The needed software, which comprises firmware of the device, Windows device driver, the dynamic link library (DLL) with API interface and in some cases even the end user applications (readily done pedagogical experiments) are or will be all available to you at no charge. The schematics of the devices are also published and if you are familiar with electronics you just need to buy specified parts and assemble the system by yourself.

Further, you may tweak or tune the system according to your needs. Say, the schematic specifies a certain operational amplifier, which produces too much noise for your particular case. You can freely swap the device with a more suitable one of your choice. With commercial systems costing e.g. 500 € or more you do not want to do that in order not to break anything or lose the warranty.

## 2.1 The choice of microcontroller and connection to a host PC

Currently, all eProDas devices are developed around the PIC18F4550 (referred to as PIC herein after) microcontroller from Microchip (www.microchip.com). The selected chip is fairly affordable since it costs around 10 € when buying in small quantities (in Slovenia). It offers a 12 MIPS microprocessor, 32 KB of program RAM, 2 KB of data RAM, 256 B of integrated EEPROM, full speed USB device functionality, one 10-bit AD converter with 13 multiplexed input channels, two voltage comparators, settable comparator reference, fairly powerful PWM module, 4 timers, power management, In-Circuit debugging and many other features that are less important for data acquisition.

The eProDas system connects to a host PC through a USB cable. At the very beginning we discarded other options such as Centronics or RS-232 connections since many new computer systems especially laptops do not support them any more. We also avoided internal connections such as PCI or AGP busses since the eProDas device would be more expensive and much less convenient to use. Namely, many pupils do not want or their parents do not let them dig into a more or less expensive personal computer with a chance to break something. Also, when you want to do an experiment like a presentation on a laptop somewhere out of your facilities you depend on laptop computers that do not have expansion slots for insertion of new devices. In such cases external connections are the only choice.

## 2.2 The eProDas device stack

In order for your eProDas device to send measured data to your application, which is running on a host PC, and receive excitation functions from it, many functional layers must cooperate as it is illustrated in the Figure 1. We informally denote the presented hierarchy the eProDas device stack.

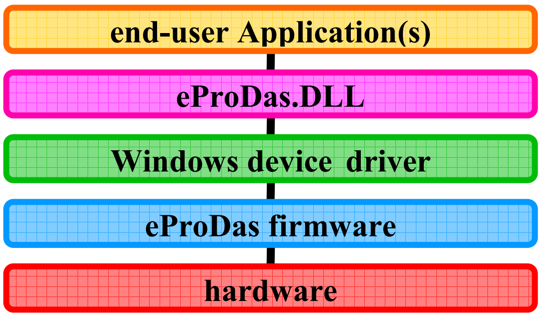


**Figure 1: eProDas device stack.**

1. At the very bottom of the stack there is electronics or hardware layer. Obviously, the hardware needs to be capable of data acquisition if you want to do one. This layer comprises of modules such as AD converters, DA converters, voltage comparators, operational amplifiers, instrumentation amplifiers, etc. The last but not the least there should be a microprocessor or microcontroller that controls your device, a suitable power supply and circuitry to physically exchange bits between eProDas device and a host PC. Luckily, our PIC possesses a lot of the just counted features and in simple cases this integrated circuit is almost all that you need.

   Every piece of hardware costs money, therefore it cannot be freely distributable by us. If you want to build your eProDas device, you have to buy its parts by yourself and assemble them properly. Within this limitation we did our best to make eProDas device as affordable as possible. First, the selected microcontroller is relatively cheap and it already contains some hardware modules that can be exploited for data acquisition purposes. Second, all necessary circuitry for establishing full speed USB connection with a host PC is already contained in a chip and is therefore included in the initial price. Third, in some cases all power that eProDas device needs is supplied by USB cable so no external power supplies are needed, which leads to further cost reduction.

2. The microcontroller needs a piece of software that tells it what we want from the device and how to control it. This software, which is located in the on-chip memory, is called firmware. It is available to you as a free-of-charge download at our web site.

3. When a device connects to a PC through USB cable Windows requires another piece of software, a device driver. Windows installation disk already contains device drivers for many categories of devices, but data acquisition systems are generally too specific to be covered by generic drivers, which means that eProDas device requires its own special driver, that is again available to you as a free-of-charge download at our web site.

4. The functionality of eProDas device is accessible to end-user applications through a set of API (application programming interface) functions that are implemented in a dynamically linked library (DLL), which (you may have guessed ☺) is again freely downloadable at our web site.

5. The end users of the system experience the eProDas devices through one or more applications. These are, for example, readily made pedagogical experiments that can be freely downloaded at our web site or they may be your own applications, which are tailored toward your specific needs.

# 3 eProDas hardware

The plain name eProDas does not denote one concrete data-acquisition device but rather a whole family of devices, which are developed around certain common denominator. These devices are then intended to cover different data-acquisition situations, by providing a suitable spectrum of data-acquisition capabilities. Some of the devices are developed by us (the eProDas developers' team), however our primary intend is to provide a solid eProDas platform as a basis for your own creative designs, since we cannot possibly fulfil each and every demand that may arise in practice.

## 3.1 The eProDas platform

The heart of all eProDas devices is the already mentioned microcontroller PIC18F4550 from Microchip, for which the pin diagram is presented in the Figure 2. Actually, there exists more than one flavour of this integrated circuit and the picture presents only the so-called "40-Pin PDIP" version of the housing. This particular choice is suitable for developing systems on a breadboard, but it has some drawbacks like the large size of the chip and the absence of the special port for In-Circuit debugging. If you are experienced electrician you will have no trouble choosing the most appropriate version of the chip for your specific needs. For the less experienced users we recommend starting with the version of the chip in the Figure 2.

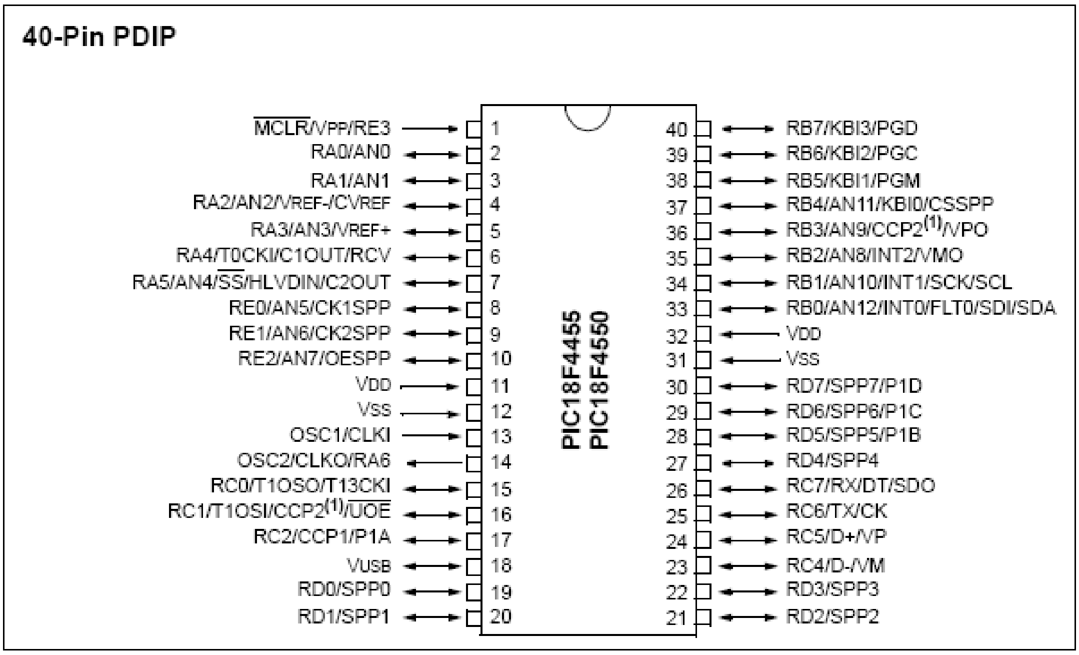


**Figure 2: Pin diagram of microcontroller PIC18F4550 (40-Pin PDIP).**

You do not have to connect all of these pins to get the operational eProDas device. On the contrary, the eProDas platform strives to waste as little PIC's resources (which include pins) on bare eProDas functionality, like communication with a host PC. Whenever possible we left features at your exposal so that you are limited only with your imagination (and with limitations of the chip, of course ☹).

Note that there also exists 28-pin version of the chip (it has no pins from RD0…RD7 and RE0…RE2), which carries the name PIC18F2550. You can freely use this model instead of PIC18F4550 if chip's size or price is priority over pin count for your particular situation of usage.

The minimal schematic that must be realized in order to make any eProDas device operational is presented in the Figure 3. We denote this schematic as the hardware part of the eProDas platform.

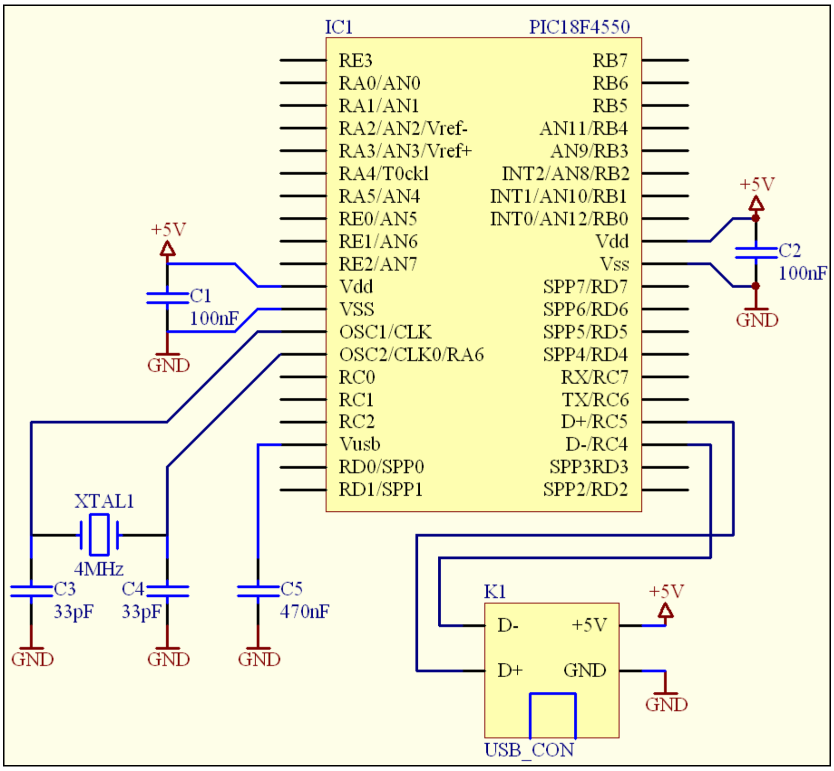


**Figure 3: The hardware part of the eProDas platform.**

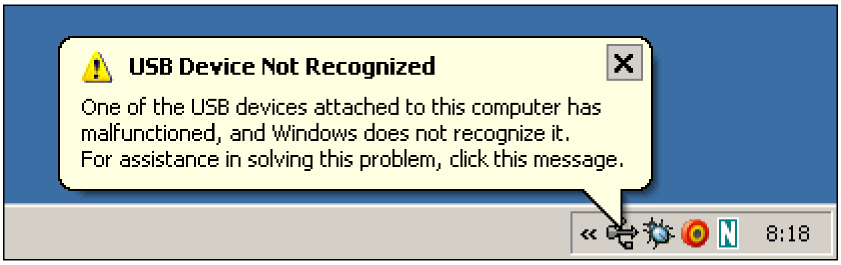
According to the schematic, the power-supply of the device is the USB cable, which can supply as much as 500 mA of current. However, bear in mind that according to the USB specifications that amount of current may be consumed only after the host PC allows it. Before the device is explicitly put into high power state, it can only consume 100 mA. Higher current consumption means that your device must possess a capability to switch the power-hungry subsystems on and off by the microcontroller. Alternatively, you may add an external 5 V regulated power supply at any time to release the burden of power supplying from the USB cable.

Let us stick with the original idea for now. The USB connector has four pins and two of them are power ones. The arrangement of the pins is specified in the lower right portion of the Figure 3, where the USB device's (B) connector is drawn. We start eProDas assembly process by taking care of properly supplying the power to the PIC. Connect both of its VSS pins (12 and 31) to the USB pin GND. Similarly, connect both VDD pins (11 and 32) to the USB pin 5 V.

In order to assure stable PIC's power and reliable device operation two blocking capacitors C1 and C2 should be connected between each pair of pins VSS and VDD as close to the chip as possible. The schematic suggests their capacitances of 100 nF, which are merely advised values and not their exact requirements. Generally, the bigger the capacitances, the more stable power supply you get, but the drawback is more in-rush current and a sharp current spike upon device connection, which should be limited to 100 mA/µs according to the USB standard. Also, do not use electrolytic or tantalum capacitors for this purpose since they are able to filter or block only relatively low frequency variations of supply voltage. If this description confuses you go to your favourite electronic shop, say out loud "I need two 100 nF blocking capacitors" and you will be fine.

You need one more blocking capacitor, which assures stable 3.3 V voltage that is needed for proper USB communication. This capacitor is annotated as C5 in the schematic and its suggested capacitance is 470 nF. Connect it between pin Vusb (18) and GND as close to the chip as possible.

If you are one of those guys that live on the edge you often underestimate the importance of blocking capacitors and drop them from your schematics. Without capacitors C1 and C2 your eProDas device may or may not work reliably, but capacitor C5 is absolutely essential. Without it Windows will disappoint you with the information balloon in the Figure 4, when you try to connect otherwise perfectly operational device to the USB cable. We warned you…



USB Device Not Recognized

One of the USB devices attached to this computer has malfunctioned, and Windows does not recognize it.
For assistance in solving this problem, click this message.

**Figure 4: The consequence of improper or missing bypassing capacitor(s).**

In addition to proper power supply bypassing your eProDas device needs an accurate and stable clock. This is obtained by utilizing quartz crystal (annotated XTAL1) of 4 MHz, which is connected between pins OSC1 (13) and OSC2 (14). You need to use exactly this frequency and you cannot temporally replace the specified crystal with some other one of a different frequency, if you do not have the proper one at hand.

For stable oscillation the crystal needs two capacitors of capacitance between 15 pF and 47 pF. In our case we use 33 pF, which works fine. However, if the constellation of your wires forms significant parasitic capacitances by itself you should lower the capacitances of capacitors. Again, make all wires as short as possible.

We are nearly at the end. All that remains to do is to connect USB differential signals, referred to as D+ and D–, to the USB connector. Do that by directly connecting PIC’s pins D+ (24) and D– (23) to the appropriate USB connector pins. One more time, make all wires as short as possible.

That’s all that it takes to build it. Do not forget to buy USB cable to connect your device to the PC.

In the Figure 5 you may examine our particular test implementation of the described schematic.

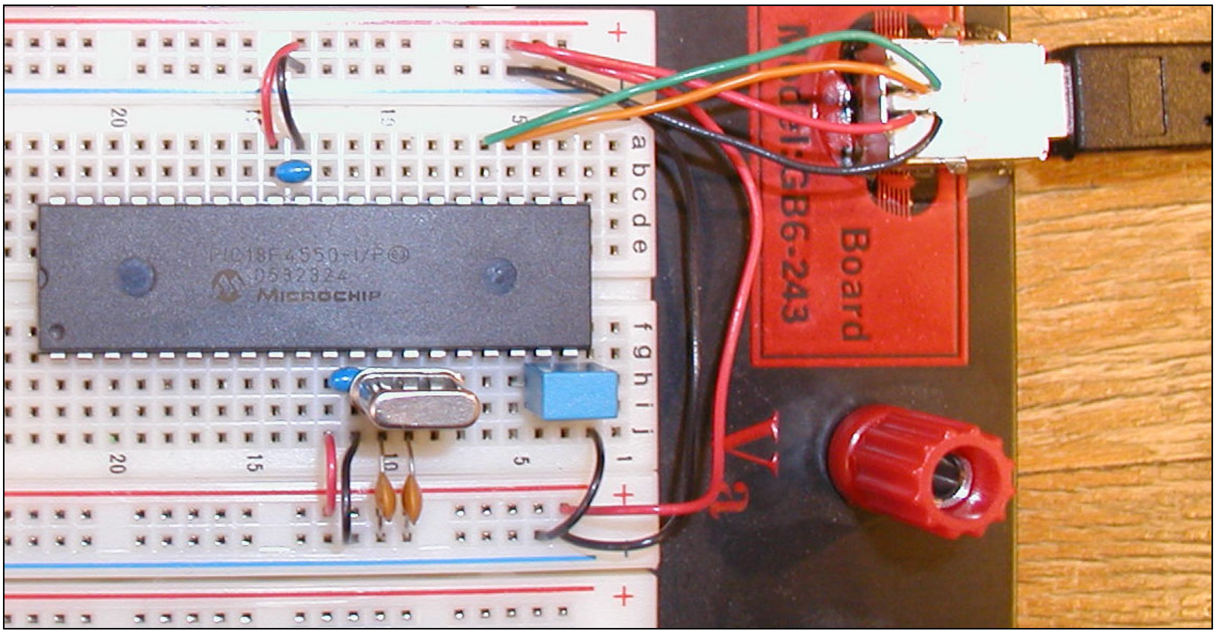


**Figure 5: An example implementation of the eProDas platform hardware.**

The figure clearly demonstrates what we mean by “…as close as possible…” and “…as short as possible…” The blocking capacitors and quartz crystal are almost touching the chip and for other parts similar conclusions may be drawn.

When you have such system at hand you can freely extend its functionality with other electronic components of your choice to do marvellous data-acquisition things with it.

# 4 Increasing efficiency of USB connection

In certain scenarios of usage you may achieve a significant performance boost of your eProDas device by taking a simple albeit costly (about 15 €) advice: do not connect eProDas device directly to a host PC but do it instead through a high speed USB 2.0 hub, as the following figure demonstrates.

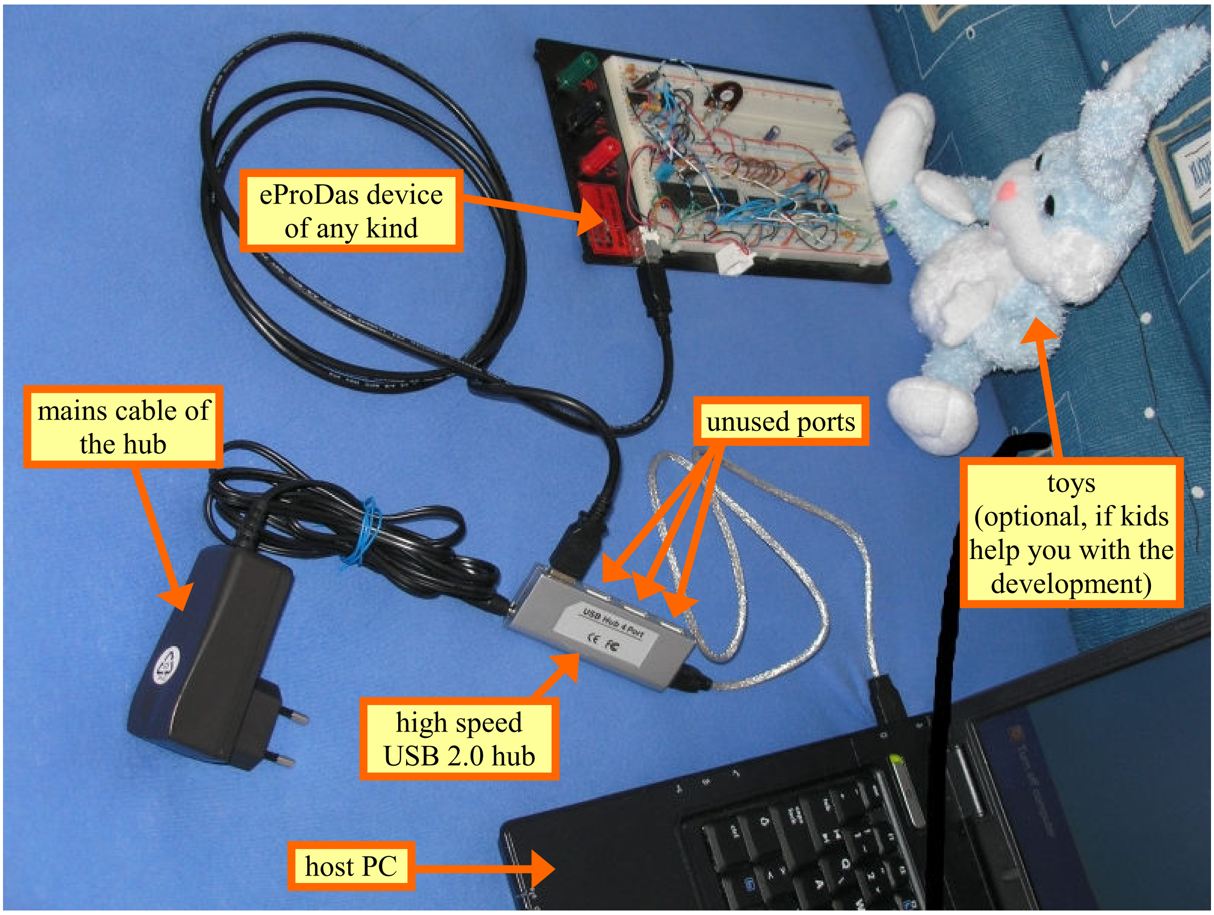

eProDas device of any kind
mains cable of the hub
unused ports
toys (optional, if kids help you with the development)
high speed USB 2.0 hub
host PC

**Figure 6: Connecting eProDas device through a high speed USB 2.0 hub.**

Our tests show that the insertion of high speed USB 2.0 hub can improve performance of *ordinary* eProDas functions (described in chapters from 15 to 18) for as much as 5 to 7 times.

eProDas also possesses the so called *periodic actions* (chapter 19). These features are designed from ground up for the speed, efficiency of execution and high data throughput. Consequently, the insertion of a hub cannot boost the performance of periodic actions as significantly as it is the case with execution of ordinary functions, although the gain may still be noticeable.

Our tests also revealed that Windows Vista works about 25 % slower than Windows XP when running on the same host PC no matter what we do.

**Please, study the following comments before rushing to your favourite computer store.**

1. Although we have tested four different hubs (from different manufacturers) and all of them performed equally well regarding the described performance increase, we cannot guarantee you that any hub on the market will give you such benefits. Please, try it by yourself and do not get mad at us if your particular piece of hardware does not work as expected, since we are giving you the advices from the bottom of our hearts to help you and to make your life happier (and no, we are not going to reveal the models of the tested hubs to you).

2. We strongly recommend you to use a self-powered hub (the one with mains cable, see the Figure 6), since in the opposite case a significant drop of the eProDas supply voltage may result. According to the USB standard, permitted range of USB provided device supply voltage may vary from 5.25 V to 4.75 V (4.4 V for low-powered hub ports) and they mean it. Host PCs should generate higher voltages than 5.0 V (but check notebooks with half empty batteries), however bus-powered hubs (the ones without mains cable) may drop these figures well below 5 V. If you intend to sample and/or generate signals within the typical voltage range from 0 V to 5 V using e.g. rail-to-rail operational amplifiers, then you know that such thing is literally impossible if the supply voltage is below 5 V. Whatever the setup you intend to use, we strongly recommend you to measure the supply voltage of the device with your favourite V-meter to make sure that the voltage is at a satisfactory level.

3. If the actual performance after the insertion of the hub is not increased as expected, try to occupy only one port of the hub (like in the Figure 6). Also, make sure that the hub that your eProDas device is connected to plugs directly to the host PC and not to another hub. Last but by no means least, it is possible that the bottleneck is not USB connection but your application, which may spend too much time manipulating its data and doing other jobs that are not directly related to communication with eProDas device. This chapter is only about increasing the raw speed of eProDas device and it says nothing about turning a 1.5 GHz CPU of your PC into 5.1 GHz one.

## Concluding remarks

You may be curious about why an addition of a simple hub increases the performance level so significantly. To tell you the truth, we would also like to know, but since our scare time has not let us study the internal working of hubs so far, we can merely guess. If you can clarify the issues or confirm our thoughts, please inform as about it. Anyway, here it follows the explanation that is a result of our bluffing skills only and not the actual expertise.

The USB protocol is packet oriented and the time axis of its eccentric world is divided into frames, which are 1 ms long in the case of full speed USB device (like eProDas is) but the length drops to eight times shorter interval of 125 μs (the so called microframe) in the case of a high speed USB device (the terminology may be misleading, if you do not know the USB standard: high speed is faster than full speed).

We strongly suspect that the host PC (this may be a flaw in USB hardware but it is more likely a problem of MS Windows) delays handling of accepted high speed bulk USB packets till the next frame occurs, which is a significant delay. However, if there is a hub along the road, then the host PC transfers the packet to the hub in high speed mode where microframes dictate the tempo. The hub by itself then delivers the packet in full speed mode to the device immediately. Therefore, the hub lowers the average latency of packet delivery from 1,000 μs to 125 μs or so, which could theoretically increase the performance for as much as eight times. This hypothesis fits well with the observations, if we take into account that commands are not processed instantly and therefore the full potential increase cannot be achieved by such simple trick.

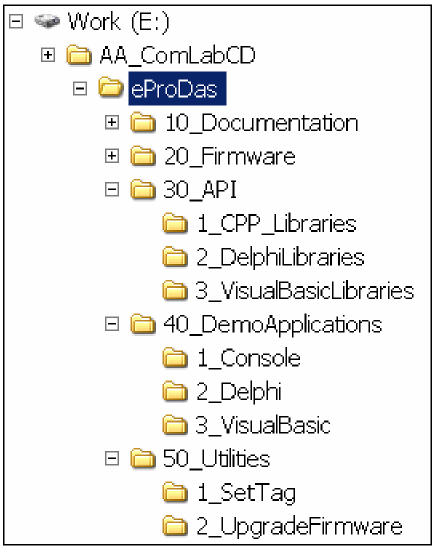
# 5 eProDas software

(Hopefully) You have successfully assembled the eProDas device according to the instructions in the chapter 3. Now you need to make it operational by putting together the needed software pieces of eProDas device stack, which can be downloaded at the following link.

*To be announced when ready. Contact us and we will send you the software by email.*

Here you can find ZIP file with the contents of the eProDas installation disk that you can extract to your hard drive or burn onto a CD/DVD, depending on your preferences.

Exploring the contents of the eProDas ZIP file will reveal the folder structure that is depicted in the Figure 7.



```
⊟ Work (E:)
    ⊞ AA_ComLabCD
    ⊟ eProDas
        ⊞ 10_Documentation
        ⊞ 20_Firmware
        ⊟ 30_API
              1_CPP_Libraries
              2_DelphiLibraries
              3_VisualBasicLibraries
        ⊟ 40_DemoApplications
              1_Console
              2_Delphi
              3_VisualBasic
        ⊟ 50_Utilities
              1_SetTag
              2_UpgradeFirmware
```

**Figure 7: The contents of the eProDas folder on the installation source.**

Please note, the drive letter `E` and the folder name `AA_ComLabCD` reflects location of installation files within our particular file system hierarchy. You are not required to place these files at the same location. Instead, extract them wherever suitable for your particular case.

The eProDas folder contains several subfolders, where the name of each of them starts with a number, which loosely suggests the order of installation or usage.

The first folder that should get your full attention is the one named `10_Documentation`. It contains the document that you are currently reading as well as other potential last minute notes about the discovered issues, which you should be familiar with. Please, examine the entire contents of this folder thoroughly and comprehensively in order to reduce the chance of disappointment with your eProDas system, since that would break our hearts.

In the following sections you will find detailed instructions on how to deal with the contents of each subfolder within the top level folder eProDas.

# 6 Firmware

The proper handling of eProDas firmware is probably the trickiest part of software installation. The challenge lies in the fact that you need a special piece of hardware, the PIC programmer, to program the firmware into the PIC microcontroller.

If you have not developed systems with microcontrollers from Microchip then you probably do not have the necessary equipment at hand. In this case you have three options. First, you can buy the needed programmer from Microchip or from some other supplier. Second, you may assemble one by yourself. Third, you may ask someone who possesses the programmer to program your PIC.

The first option will cost you around 140 € (in Slovenia) if you buy the Microchip's In-Circuit Debugger 2 (see the Figure 8), for which you can find more information at Microchip homepage.

**Figure 8: The In-Circuit Debugger 2 from Microchip.**

This piece of hardware, which by the way we use for the development of eProDas, is not only a programmer but also a true real-time debugger, as it name reveals. If you intend to play with Microchip microcontrollers for other purposes in addition to kick starting the eProDas device, the purchase may be worth your money. However, if one-time programming of PIC microcontroller is the only motive for spending the money on this equipment, it is probably not a good idea. Try to find a better solution in the following paragraphs.

If the price for the official programmer is too high for your taste, there exist many cheaper alternatives from other suppliers. Please, do not ask us for a recommendation, since we use only official Microchip equipment.

Still further cost saving may be achieved by building the required programmer by oneself according to instructions found on the Internet. A simple search by your favourite search engine will surely give you viable options. Again, do not seek the recommendation from our side, since we do not have any hands on experiences with these devices, but we know people that use them successfully. The drawback of this approach is that you have to invest some of your precious time into building the device, but the costs may be on an order of 15 €.

For the non-developers among you the best choice is to become a friend with a PIC expert in your neighbourhood. She or he will surely be kind enough to program your PIC with eProDas firmware. For an experienced person the whole process takes no more than a minute.

## 6.1 Programming the chip with the firmware

When you have a hardware programmer or a person with it at hand you may actually program the firmware into the PIC 18F4550 integrated circuit. The firmware is contained in the file eProDas_M_m.HEX, which is located in the folder `20_Firmware` of the eProDas installation disk. The version of the firmware is indicated by numbers M (major version) and m (minor version).

For example, the file name `eProDas_1_0.HEX` reveals the firmware version 1.0.

Microchip In-Circuit Debugger 2 can directly take the HEX file as the input and program your chip in a straightforward way. However, some home-made programmers need a slightly different file format than the main HEX file contains. For that matter, there exists a subfolder `OtherFormats` within the folder `20_Firmware`, which contains the firmware in different formats. By selecting a proper input file you should be able to program your PIC with and PIC programmer that you can find on the white or the black market.

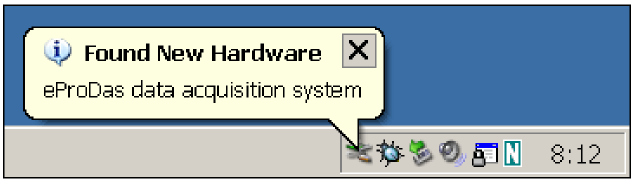
So, follow the instructions for your particular programmer and in a matter of seconds your PIC chip is not just a chip any more but a brain of your eProDas device.

# 7 Installation of device driver

Please note. Before you can proceed with this section your eProDas device has to be completely assembled and eProDas firmware needs to be programmed into the PIC microcontroller. If this is not the case please return to the previous sections and remedy the situation.

These instructions assume that you have installed Microsoft Windows XP or Microsoft Windows Vista operating system (please, see chapter 12 if this is not the case). It is also assumed that you have installed all service packs and other patches/bugfixes that are available for your operating system from “Microsoft Update” web site.

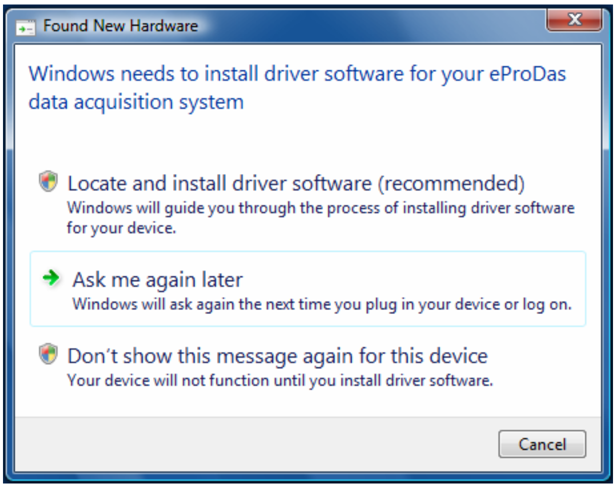
When correctly assembled and programmed eProDas device is plugged into the USB bus for the first time, Windows will discover new piece of hardware and let you know about this with “Found New Hardware” information balloon (Windows XP, Figure 10) or pop-up window (Windows Vista, Figure 10).



Found New Hardware

eProDas data acquisition system

**Figure 9: Found New Hardware information balloon (Windows XP).**



Found New Hardware

Windows needs to install driver software for your eProDas data acquisition system

Locate and install driver software (recommended)
Windows will guide you through the process of installing driver software for your device.

Ask me again later
Windows will ask again the next time you plug in your device or log on.

Don’t show this message again for this device
Your device will not function until you install driver software.

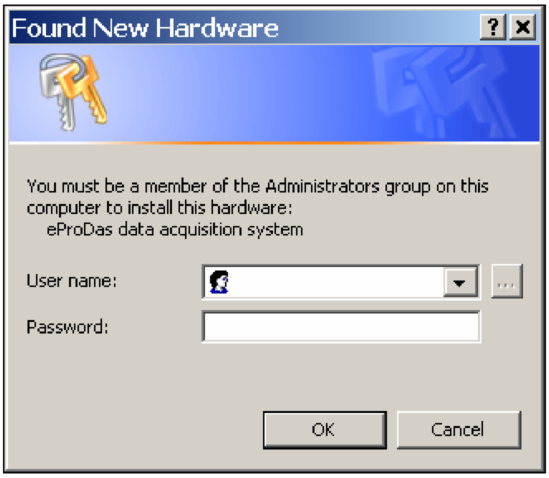
**Figure 10: Found New Hardware pop-up (Windows Vista).**

If the balloon or window does not pop up or if the message does not reveal the discovery of the “eProDas data acquisition system” your hardware or firmware is not operational and you cannot continue with the installation. Please, correct the issues according to the instructions in the previous sections and return to this step thereinafter.

With Windows Vista click the first offered option “Locate and install driver software” in the Figure 10 which brings up dialog box where you manually grant permission to proceed with installation.

In order to continue with the installation you need to be logged in to Windows as an administrator or the following dialog box in Figure 11 will appear (in the case of Windows XP), by means of which you can type in an administrator password to confirm that you are entitled to install the device driver. Windows Vista acts similarly but the dialog box is stilistically different.

Note. If you cannot obtain administrative privileges on the computer, you will not be able to continue the installation.



Found New Hardware

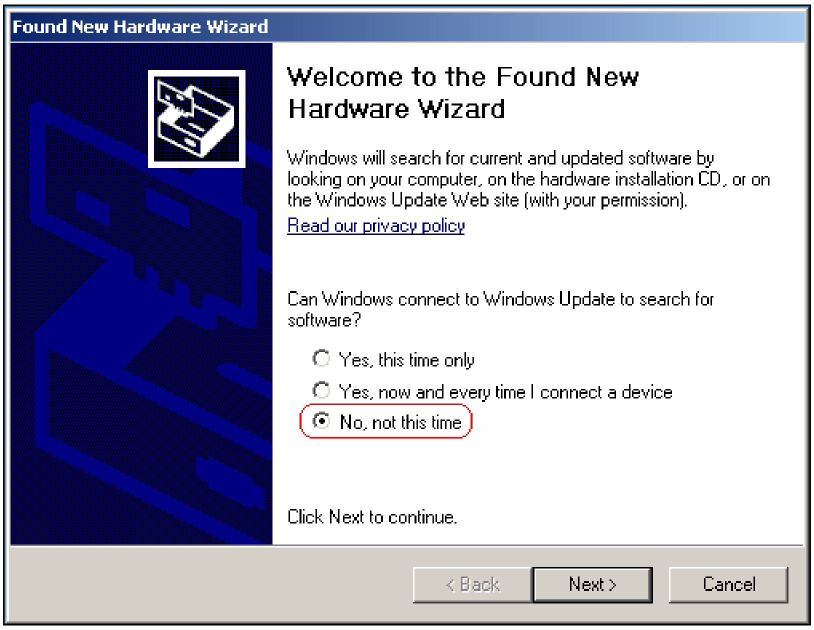
You must be a member of the Administrators group on this computer to install this hardware:
eProDas data acquisition system

User name:

Password:

**Figure 11: Obtaining the administrative privileges (Windows XP).**

Since Windows does not know anything about your eProDas device, it will inquire you about the proper driver for the device with the dialog box, shown in the Figure 12 (Windows XP) or Figure 13 (Vindows Vista).



Found New Hardware Wizard

Welcome to the Found New Hardware Wizard

Windows will search for current and updated software by looking on your computer, on the hardware installation CD, or on the Windows Update Web site (with your permission).

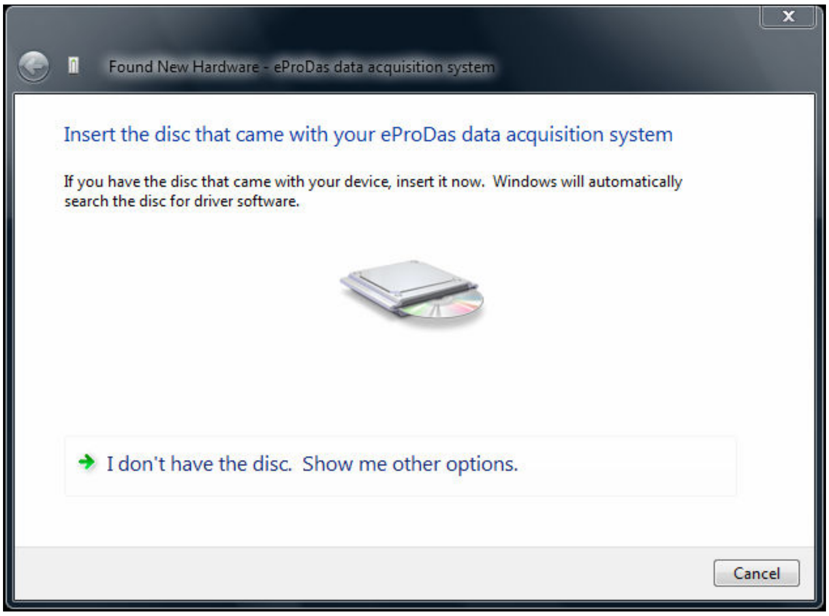
Read our privacy policy

Can Windows connect to Windows Update to search for software?

○ Yes, this time only
○ Yes, now and every time I connect a device
● No, not this time

Click Next to continue.

**Figure 12: The first screen of the New Hardware Wizard (Windows XP).**



Found New Hardware - eProDas data acquisition system

Insert the disc that came with your eProDas data acquisition system

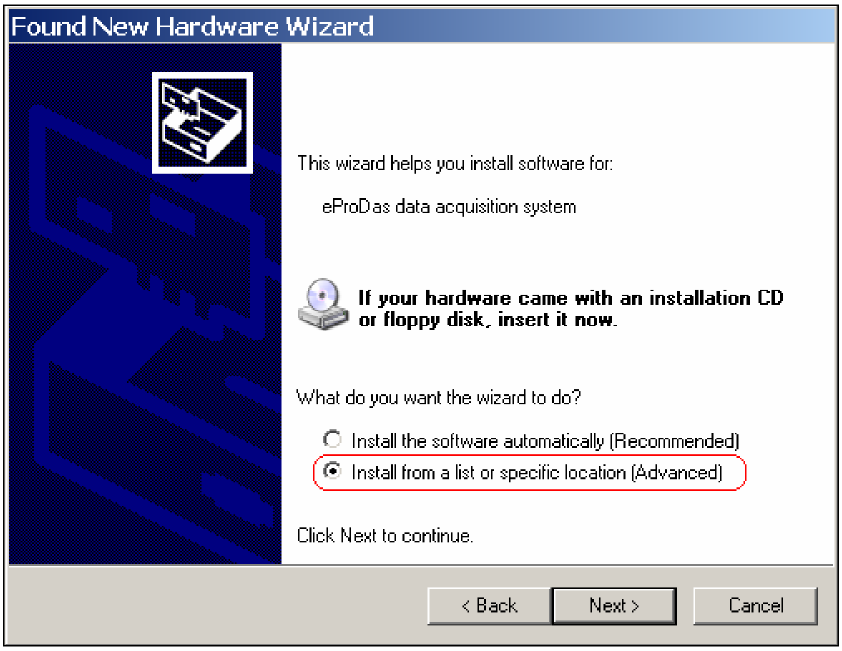
If you have the disc that came with your device, insert it now. Windows will automatically search the disc for driver software.

I don't have the disc. Show me other options.

**Figure 13: The first screen of the New Hardware Wizard (Windows Vista).**

In Windows XP click on “No, not this time”, as suggested by Figure 12. In Windows Vista click on “I don’t have the disc. Show...”.

In the next dialog box select “Advanced” option as instructed by the Figure 14 (Windows XP) or select “Browse my computer for driver software” (Figure 15, Windows Vista).



Found New Hardware Wizard

This wizard helps you install software for:

eProDas data acquisition system

**If your hardware came with an installation CD or floppy disk, insert it now.**

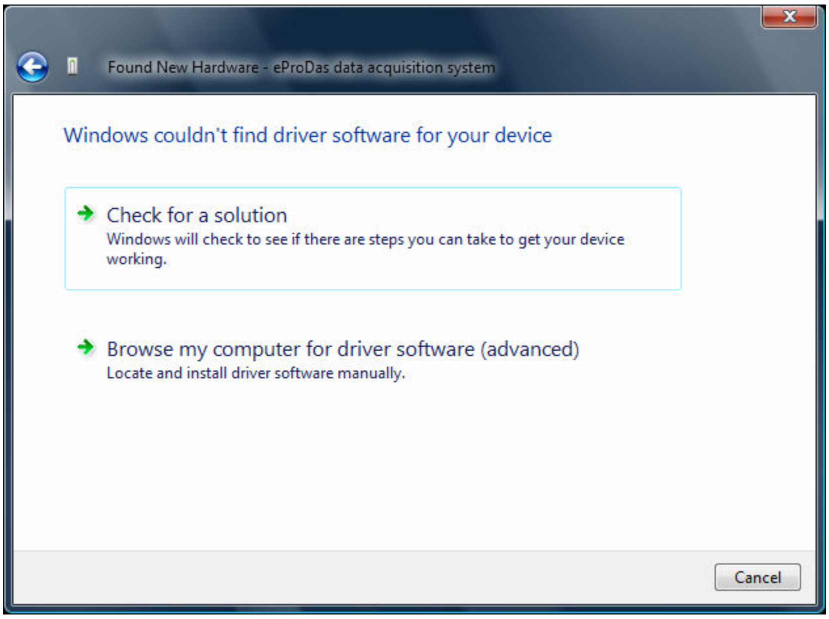
What do you want the wizard to do?

○ Install the software automatically (Recommended)

● Install from a list or specific location (Advanced)

Click Next to continue.

**Figure 14: Automatic or manual searching for the driver (Windows XP).**



Found New Hardware - eProDas data acquisition system

Windows couldn't find driver software for your device

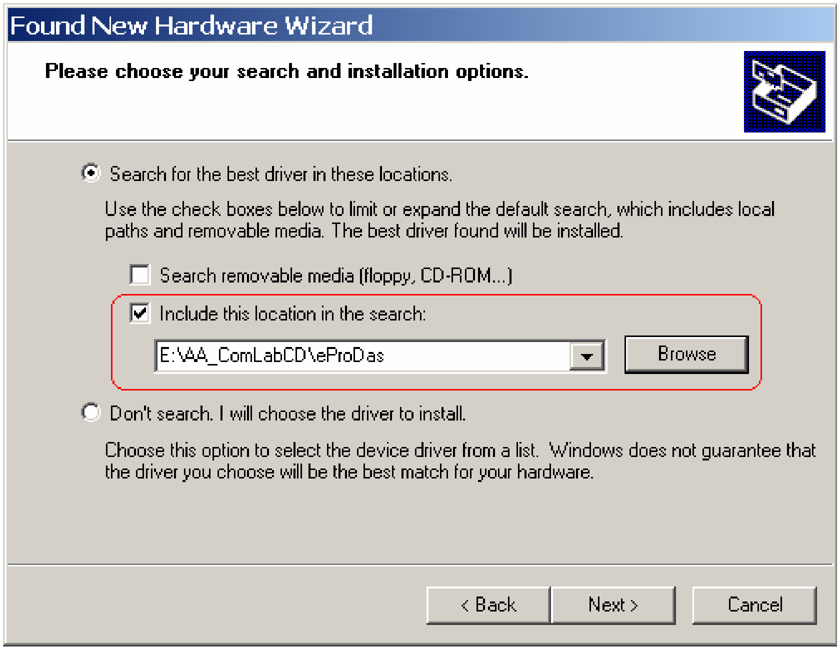
Check for a solution
Windows will check to see if there are steps you can take to get your device working.

Browse my computer for driver software (advanced)
Locate and install driver software manually.

**Figure 15: Selection of "Browse for driver" (Windows Vista).**

In Windows XP select "No, not this time" when being asked whether to search for the proper driver on the Internet and click Next. In the next step (the Figure 16) click Browse and point Windows to the root folder of eProDas installation; this folder contains files `eProDas.INF`, `eProDas.ID`, `eProDas.DLL` and `eProDas.sys` (and possibly other ones).

In the case of Windows Vista perform the same action with differently polished dialog boxes.



Found New Hardware Wizard

Please choose your search and installation options.

● Search for the best driver in these locations.

Use the check boxes below to limit or expand the default search, which includes local paths and removable media. The best driver found will be installed.

□ Search removable media (floppy, CD-ROM...)

☑ Include this location in the search:

E:\AA_ComLabCD\eProDas

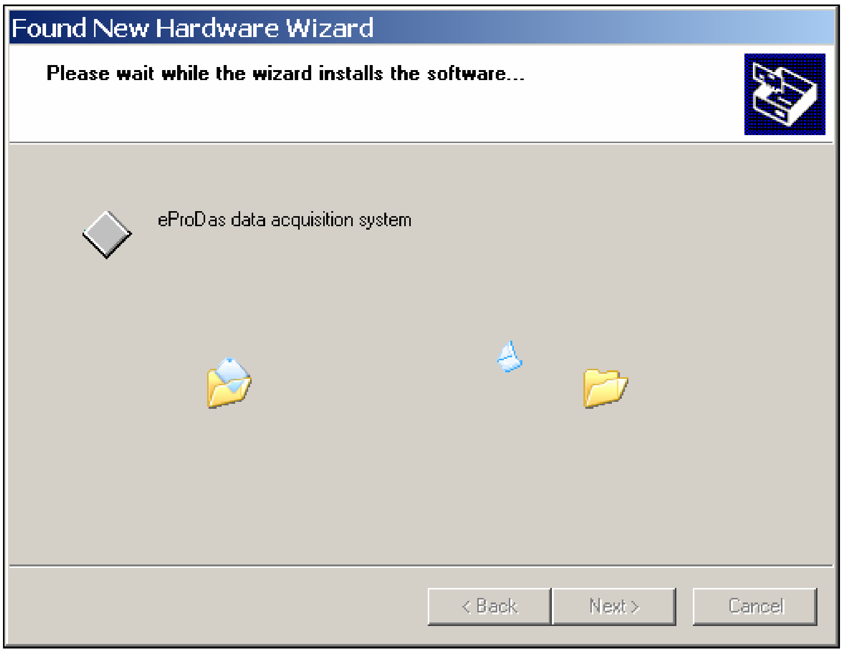
○ Don't search. I will choose the driver to install.

Choose this option to select the device driver from a list. Windows does not guarantee that the driver you choose will be the best match for your hardware.

**Figure 16: Specifying location of the driver (Windows XP).**

Please note that the displayed content `E:\AA_ComLabCD\eProDas` is a proper entry for our particular case only. Your specification must reflect the actual location of the ComLab software on your disk.

Click Next when you are done. If all is well, Windows XP will install the driver on your system. During the process you will notice the dialog box in the Figure 17.



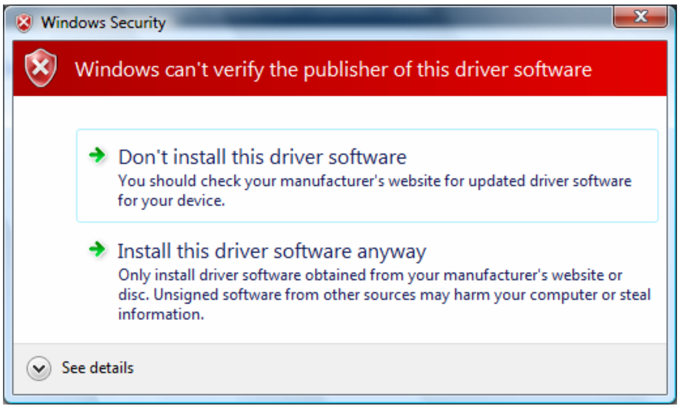
Found New Hardware Wizard

Please wait while the wizard installs the software...

eProDas data acquisition system

**Figure 17: Driver installation by Windows XP.**

If you are running Windows Vista, you need to comfirm installation since eProDas driver is not officially signed. For that matter the dialog box in Figure 18 will appear. Select “Install this driver software anyway”.



Windows Security

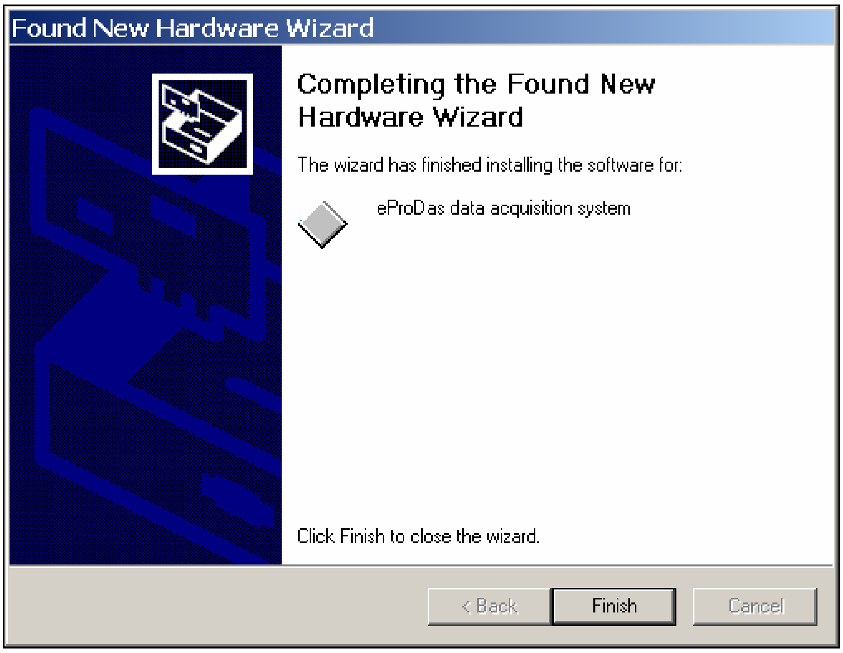
Windows can't verify the publisher of this driver software

Don't install this driver software
You should check your manufacturer's website for updated driver software for your device.

Install this driver software anyway
Only install driver software obtained from your manufacturer's website or disc. Unsigned software from other sources may harm your computer or steal information.

**Figure 18: Comfirmation of driver installation in Windows Vista.**

When Windows successfully finishes installing the driver it will delight you with the following dialog.



Found New Hardware Wizard

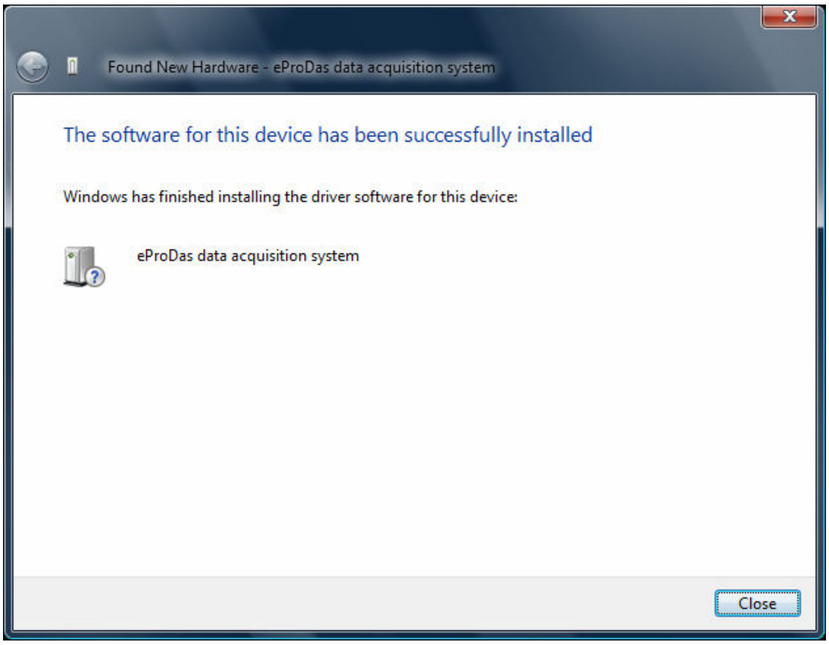
Completing the Found New Hardware Wizard

The wizard has finished installing the software for:

eProDas data acquisition system

Click Finish to close the wizard.

**Figure 19: The end of the successful installation (Windows XP).**



Found New Hardware - eProDas data acquisition system

The software for this device has been successfully installed

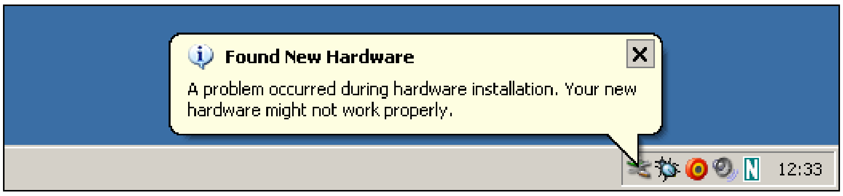
Windows has finished installing the driver software for this device:

eProDas data acquisition system

**Figure 20: The end of the successful installation (Windows Vista).**

That’s all, folks. Click Finish (XP) or Close (Vista) and proceed with the next section.

In the case that you aborted the installation or if something went wrong during the process, the following balloon (Windows XP, similar message in Windows Vista) will inform you that eProDas device is not readily setup and you cannot use it at the moment.



Found New Hardware

A problem occurred during hardware installation. Your new hardware might not work properly.

**Figure 21: The end of an unsuccessful installation (Windows XP).**

To correct the issues simply disconnect your eProDas device from the USB cable and reconnect it at a later time when you are ready to complete the installation.

## Checking the installation

This section is not required to be complied with. However, if you are not sure whether the installation process was completed flawlessly or if you want to check you eProDas system for whatever reason, you may follow the guidelines. The eProDas device needs to be connected all the time during the process of checking.

The following description and screen shots are in accordance with Windows XP. A similar procedure applies to Windows Vista also.

Right click on the `My Computer` icon to get the pop-up menu in the Figure 22.

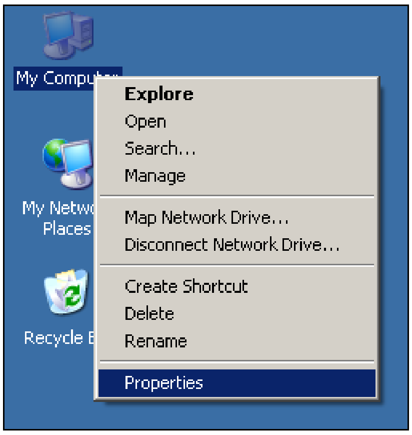
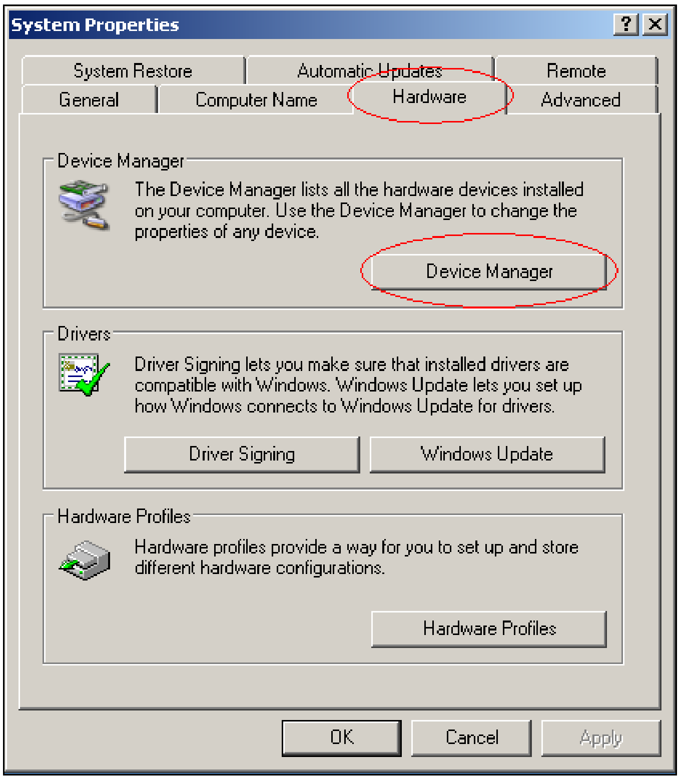


**Figure 22: Pop-up menu of My Computer.**

Click on `Properties` to get the following dialog box. Select `Hardware` tab and click `Device Manager`.



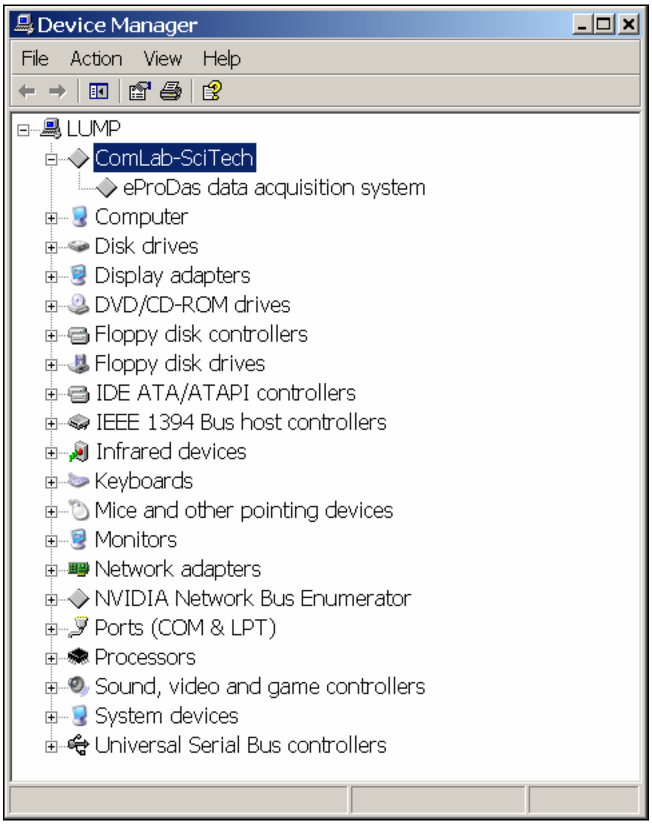
The Device Manager lists all the hardware devices installed on your computer. Use the Device Manager to change the properties of any device.

Driver Signing lets you make sure that installed drivers are compatible with Windows. Windows Update lets you set up how Windows connects to Windows Update for drivers.

Hardware profiles provide a way for you to set up and store different hardware configurations.

**Figure 23: System properties dialog box.**

This will bring you the list of all (non-hidden) devices on your system. Within the list there should be a category `ComLab-SciTech`.



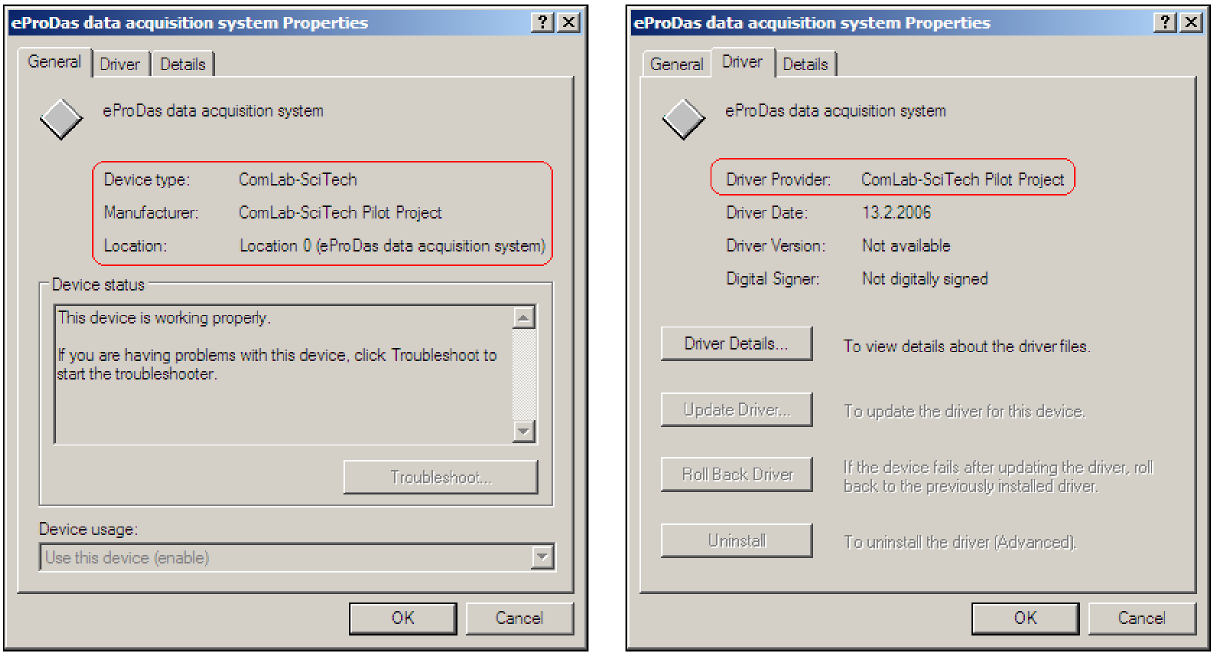
```
LUMP
  ComLab-SciTech
    eProDas data acquisition system
  Computer
  Disk drives
  Display adapters
  DVD/CD-ROM drives
  Floppy disk controllers
  Floppy disk drives
  IDE ATA/ATAPI controllers
  IEEE 1394 Bus host controllers
  Infrared devices
  Keyboards
  Mice and other pointing devices
  Monitors
  Network adapters
  NVIDIA Network Bus Enumerator
  Ports (COM & LPT)
  Processors
  Sound, video and game controllers
  System devices
  Universal Serial Bus controllers
```

**Figure 24: The list of all devices in the system.**

**Note.** Depending on the configuration of your system, the actual list on your screen will certainly be different than the one in the Figure 24. All that matters is that the category `ComLab-SciTech` does exist somewhere on the list.

Click on the small cross to the left of the category to expand the group and spot the `eProDas data acquisition system` entry. Right click on it to get the associated pop-up menu and select properties.

Windows will summarize the `Device Type`, its `Manufacturer` and confirm the name of the device one more time in the parentheses next to the `Location` label as presented in the Figure 25 (left).



ePro Das data acquisition system

Device type: ComLab-SciTech
Manufacturer: ComLab-SciTech Pilot Project
Location: Location 0 (ePro Das data acquisition system)

Device status

This device is working properly.

If you are having problems with this device, click Troubleshoot to start the troubleshooter.

Device usage: Use this device (enable)

ePro Das data acquisition system

Driver Provider: ComLab-SciTech Pilot Project
Driver Date: 13.2.2006
Driver Version: Not available
Digital Signer: Not digitally signed

Driver Details... To view details about the driver files.

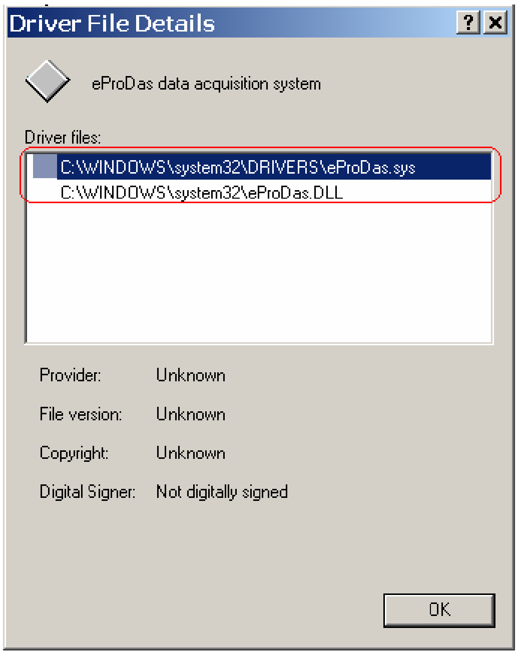
Update Driver... To update the driver for this device.

Roll Back Driver If the device fails after updating the driver, roll back to the previously installed driver.

Uninstall To uninstall the driver (Advanced).

**Figure 25: General tab of eProDas properties (left) and its Driver tab (right).**

Under the `Driver` tab check the `Driver Provider` which should look like the one in the Figure 25 (right). Next, click the `Driver Details…` button to get the information in the Figure 26 (left).



ePro Das data acquisition system

Driver files:

C:\WINDOWS\system32\DRIVERS\eProDas.sys
C:\WINDOWS\system32\eProDas.DLL

Provider: Unknown
File version: Unknown
Copyright: Unknown
Digital Signer: Not digitally signed

**Figure 26: Driver File Details of eProDas (left) and Details tab of device driver (right).**

Verify that the files `eProDas.sys` and `eProDas.DLL` were copied into the folders `Windows\system32\DRIVERS` and `Windows\system32`, respectively.

# 8 Console application

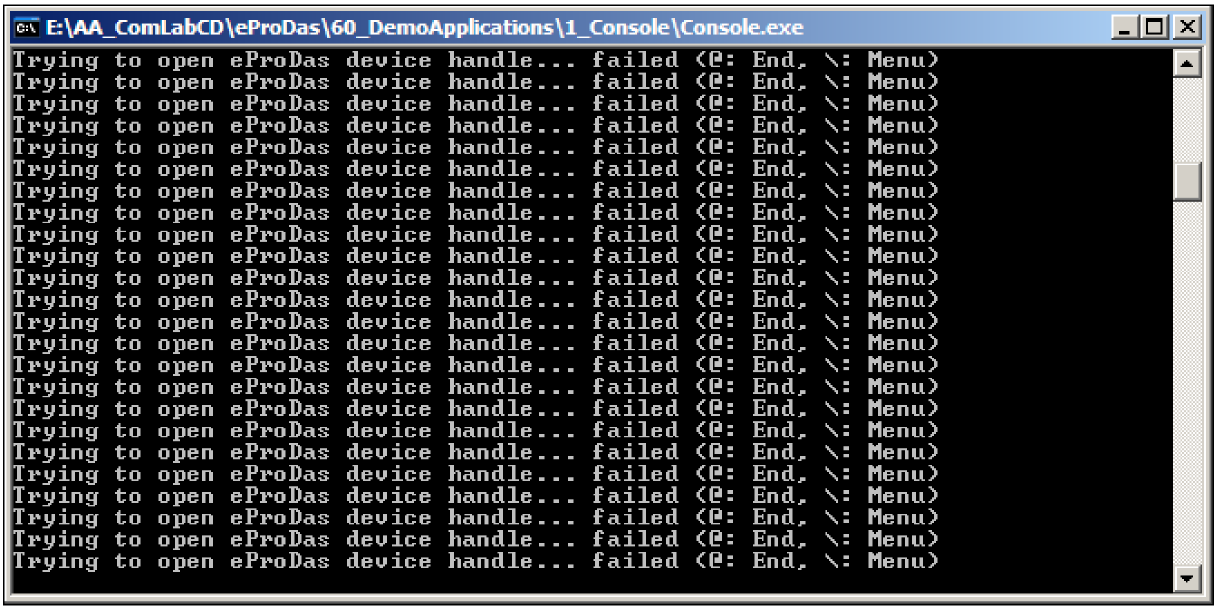
In the folder `eProDas/40_DemoApplications/1_Console` you can find a simple eProDas console application. This small piece of code serves two purposes. First, it is our diagnostic tool, which is developed in parallel with eProDas stack. It enables us to test the eProDas system without wasting our time on GUI programming. Although it is not particularly user friendly program it may be of a value to you too. Namely, the configuration of eProDas device and states of its input/output pins can be monitored in real time for performing simple circuit diagnostics.

Second, the developers among you may examine the source code of console application to see how functions in `eProDas.DLL` are utilized. However, bear in mind that being diagnostic utility this program is not written in the same style as true eProDas applications are expected to be. For example, after you configure the eProDas device to suit a certain purpose, you do not need to check its configuration (if you know what you are doing :-), but this program does it in order to summarize the up to date configuration on the screen. Also, there are many code fragments to report other information, such as version of `eProDas.DLL` and Firmware, the last RESET type etc. You, the application developer, may ignore all these non-relevant parts and concentrate only on the flesh and blood of your masterpieces.

The usage of console application is not going to be described in detail, since we believe it is simple enough for anyone to figure out how to deal with it. However, some tips for efficient use are worth to mention.

## 8.1 Running the console application for the first time

When you run the `Console.EXE` application you will probably see something like the Figure 27.



E:\AA_ComLabCD\eProDas\60_DemoApplications\1_Console\Console.exe

```
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
```

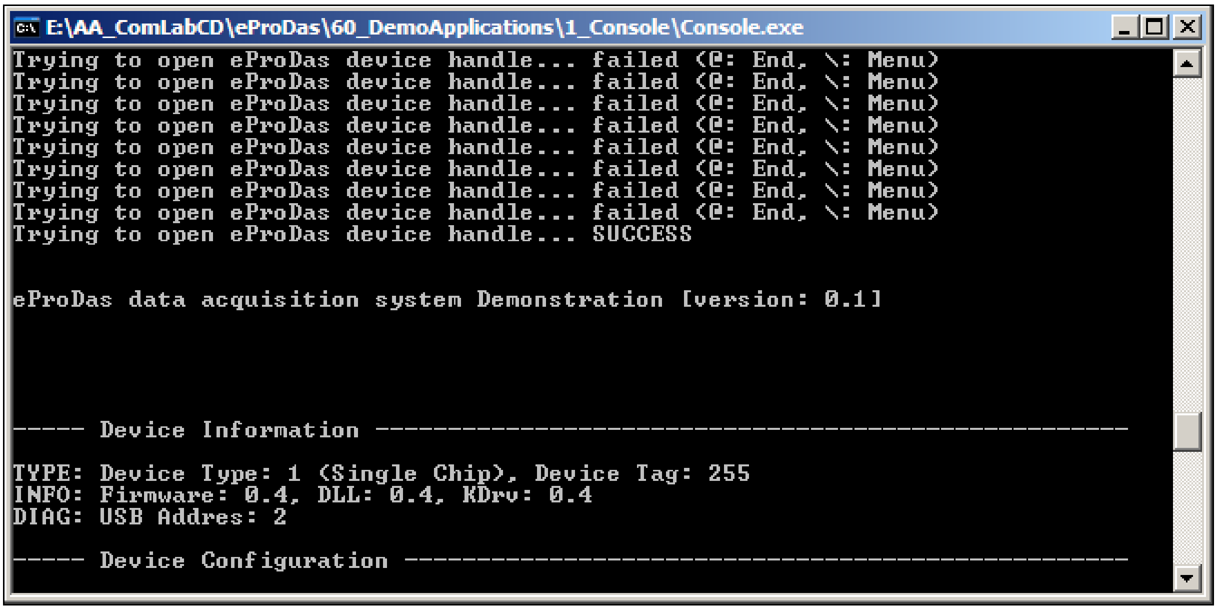
**Figure 27: Typical first-time console window.**

**Note.** In the course of development the actual output may change and this and the following figures may not accurately reflect the contents of your screen.

For your convenience the console application is automatically trying to establish the connection with your eProDas device by opening the appropriate device handle (see the programming manual later on for a thorough description). If the eProDas device is not connected to the USB bus or worse still if it is not working properly, the attempt would fail with the information message “failed”. The console application repeats the attempt forever until you instruct it to stop, which explains the repetition of displayed lines in the Figure 27.

If you can remedy the problem and put the eProDas device in operational state, simply do it. Otherwise, press “@” to exit from the application or backslash “\” to skip establishing the connection and jump directly to the application’s main menu where all commands are at your exposal. If you try to execute anything without the operational eProDas device the attempt will fail for sure.

In our case, the problem was due to the disconnected eProDas device from the USB cable. A second after we plugged it in the socket, the screen changed to something in the Figure 28.


E:\AA_ComLabCD\eProDas\60_DemoApplications\1_Console\Console.exe

```
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... SUCCESS


eProDas data acquisition system Demonstration [version: 0.1]




----- Device Information ----------------------------------------------------

TYPE: Device Type: 1 (Single Chip), Device Tag: 255
INFO: Firmware: 0.4, DLL: 0.4, KDrv: 0.4
DIAG: USB Addres: 2

----- Device Configuration --------------------------------------------------
```

**Figure 28: Successful establishment of connection with the eProDas device.**

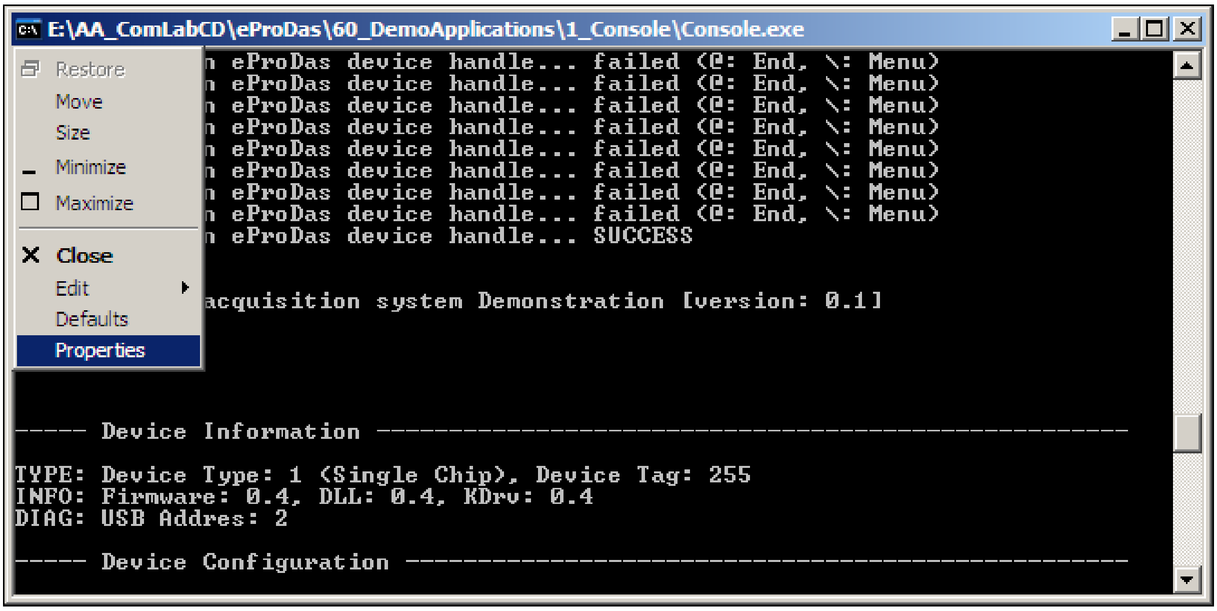
The console application starts to print device and configuration information that it obtains by querying the device through API functions in `eProDas.DLL`.

As any useful diagnostics tool this program intends to present as much information about the eProDas device to the user as possible. This means that it is “designed” for producing big chunks of textual outputs. In fact, you will not have a chance to see the contents of the Figure 28, since it is going to be overwritten by the additional information that follows.

If you intend to play with this application, we strongly recommend you to increase the size of the console window from default 80x25 characters to a more appropriate size. In case you do not know how to do it, please follow the steps in the next section.

## 8.2 Expanding the console window

Click anywhere in the title line with the right mouse button to a the pop-up menu, as it is seen in the Figure 29.

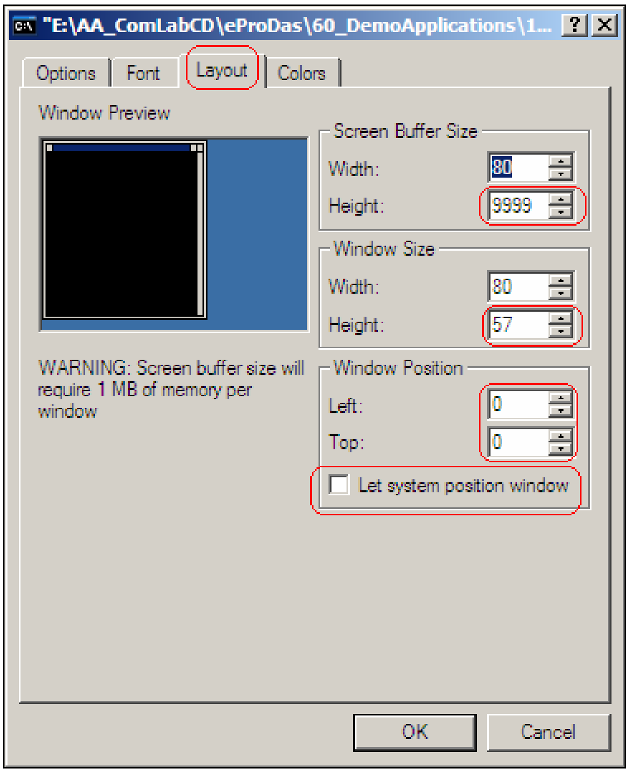


E:\AA_ComLabCD\eProDas\60_DemoApplications\1_Console\Console.exe

```
n eProDas device handle... failed (@: End, \: Menu)
n eProDas device handle... failed (@: End, \: Menu)
n eProDas device handle... failed (@: End, \: Menu)
n eProDas device handle... failed (@: End, \: Menu)
n eProDas device handle... failed (@: End, \: Menu)
n eProDas device handle... failed (@: End, \: Menu)
n eProDas device handle... failed (@: End, \: Menu)
n eProDas device handle... failed (@: End, \: Menu)
n eProDas device handle... SUCCESS

acquisition system Demonstration [version: 0.1]




----- Device Information ----------------------------------------------------

TYPE: Device Type: 1 (Single Chip), Device Tag: 255
INFO: Firmware: 0.4, DLL: 0.4, KDrv: 0.4
DIAG: USB Addres: 2

----- Device Configuration --------------------------------------------------
```

**Figure 29: The pop-up menu of the console window.**

Click on Properties with the left button to arrive at the dialog box in Figure 30.



"E:\AA_ComLabCD\eProDas\60_DemoApplications\1...

Screen Buffer Size
Width: 80
Height: 9999

Window Size
Width: 80
Height: 57

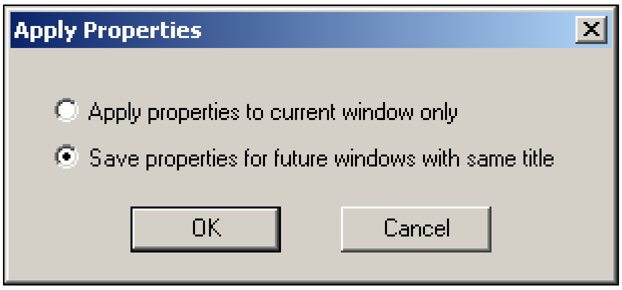
Window Position
Left: 0
Top: 0

□ Let system position window

WARNING: Screen buffer size will require 1 MB of memory per window

**Figure 30: Dialog box for altering the console window settings.**

The default width of 80 is OK for the screen buffer size as well as for the window size. We recommend to you to increase the Height of the Screen buffer size to some large value, which will enable you to scroll the contents of the console window backward to examine the currently overwritten contents. Also, for greater pleasure of working with the application increase the window size to at least 40 or so. If screen resolution and the choice of font permits you even bigger window configure it freely that way. In addition to bigger screen we also appreciate that the window is always placed at the same position on screen, which you may instruct by altering the settings in the Window Position group of controls. When you click OK, Windows will prompt you with dialog box in the next figure.



Apply Properties

○ Apply properties to current window only

● Save properties for future windows with same title

**Figure 31: The choice of temporal or permanent new settings.**

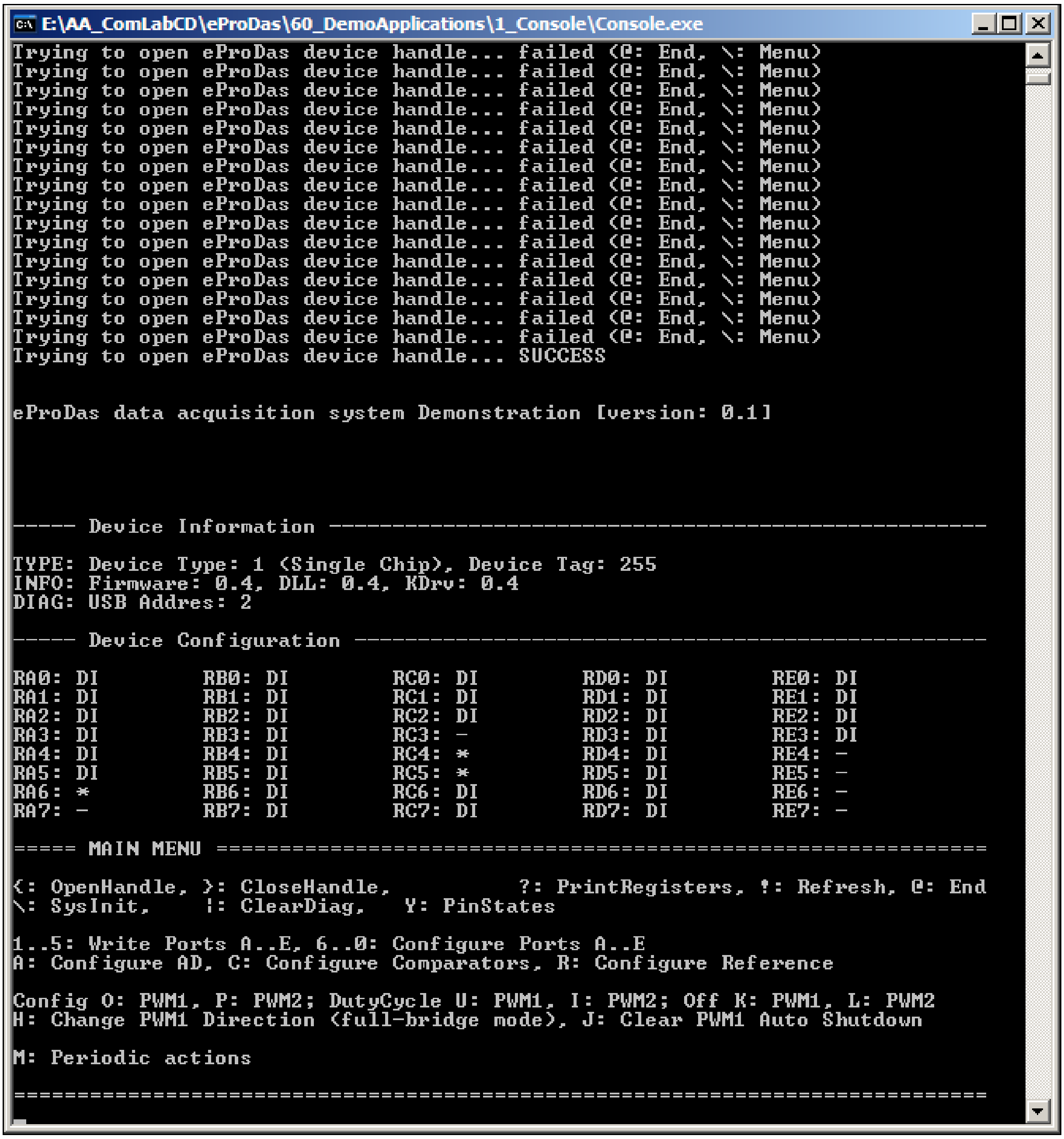
Select “Save properties for future windows with the same title” in order to make the changes permanent. After clicking OK, you may start using the console application in a more comfortable way.

## 8.3 Using the console application

The Figure 32 presents the more complete contents of the Figure 28. Under the `Device Information` section you can find the type of your device and its tag (explained in section 10.3), version numbers of various pieces of the eProDas device stack (Firmware, DLL, Windows device driver), and other information.

Under DIAG(nostics) section, the USB address of the device that is assigned to it by Windows is displayed. The RESET type, if different from the Power-On reset, and firmware upgrade information is also summarized when appropriate. If eProDas device finds any errors during operation, these are also listed here.

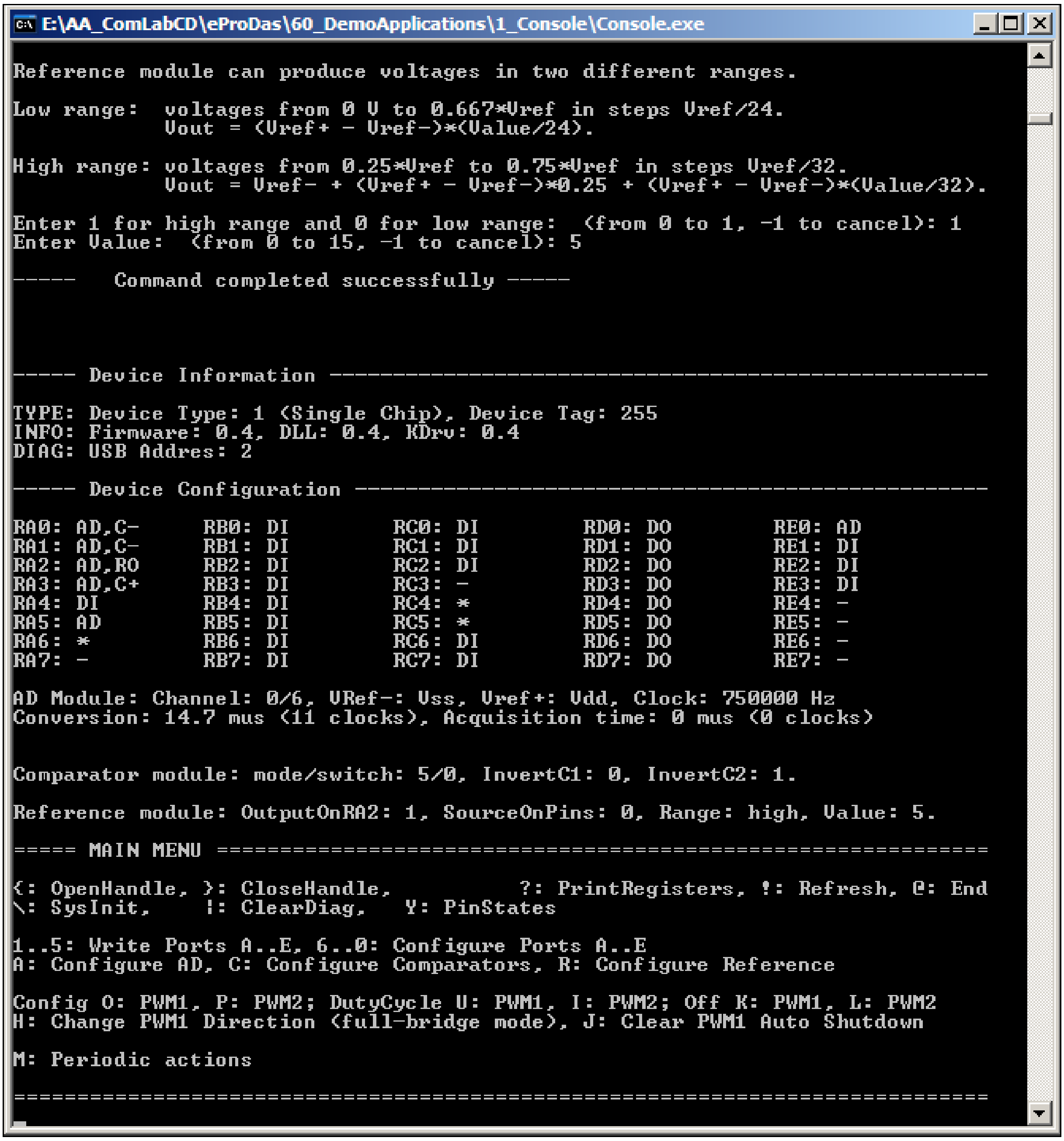
`Device Configuration` section displays more useful information for the end user of the system. Here you will find the configuration of each available general purpose pin of the PIC microcontroller. The comprehensive explanation of pin annotations is described in section 25.1.3. However, for your convenience we mention here that `DI` means digital input, whereas the marks “-” and “*” denote non-existing and reserved pin, respectively.


E:\AA_ComLabCD\eProDas\60_DemoApplications\1_Console\Console.exe

```
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... failed (@: End, \: Menu)
Trying to open eProDas device handle... SUCCESS


eProDas data acquisition system Demonstration [version: 0.1]





----- Device Information ----------------------------------------------------

TYPE: Device Type: 1 (Single Chip), Device Tag: 255
INFO: Firmware: 0.4, DLL: 0.4, KDrv: 0.4
DIAG: USB Addres: 2

----- Device Configuration --------------------------------------------------

RA0: DI        RB0: DI        RC0: DI        RD0: DI        RE0: DI
RA1: DI        RB1: DI        RC1: DI        RD1: DI        RE1: DI
RA2: DI        RB2: DI        RC2: DI        RD2: DI        RE2: DI
RA3: DI        RB3: DI        RC3: -         RD3: DI        RE3: DI
RA4: DI        RB4: DI        RC4: *         RD4: DI        RE4: -
RA5: DI        RB5: DI        RC5: *         RD5: DI        RE5: -
RA6: *         RB6: DI        RC6: DI        RD6: DI        RE6: -
RA7: -         RB7: DI        RC7: DI        RD7: DI        RE7: -

===== MAIN MENU =============================================================

<: OpenHandle, >: CloseHandle,          ?: PrintRegisters, !: Refresh, @: End
\: SysInit,    |: ClearDiag,   Y: PinStates

1..5: Write Ports A..E, 6..0: Configure Ports A..E
A: Configure AD, C: Configure Comparators, R: Configure Reference

Config O: PWM1, P: PWM2; DutyCycle U: PWM1, I: PWM2; Off K: PWM1, L: PWM2
H: Change PWM1 Direction (full-bridge mode), J: Clear PWM1 Auto Shutdown

M: Periodic actions

=============================================================================
```

**Figure 32: The successful establishment of connection on a big screen.**

When you configure device modules, such as voltage comparators or voltage reference, additional descriptions will follow the pin table. The Figure 33 presents a more elaborate example of eProDas configuration.


E:\AA_ComLabCD\eProDas\60_DemoApplications\1_Console\Console.exe

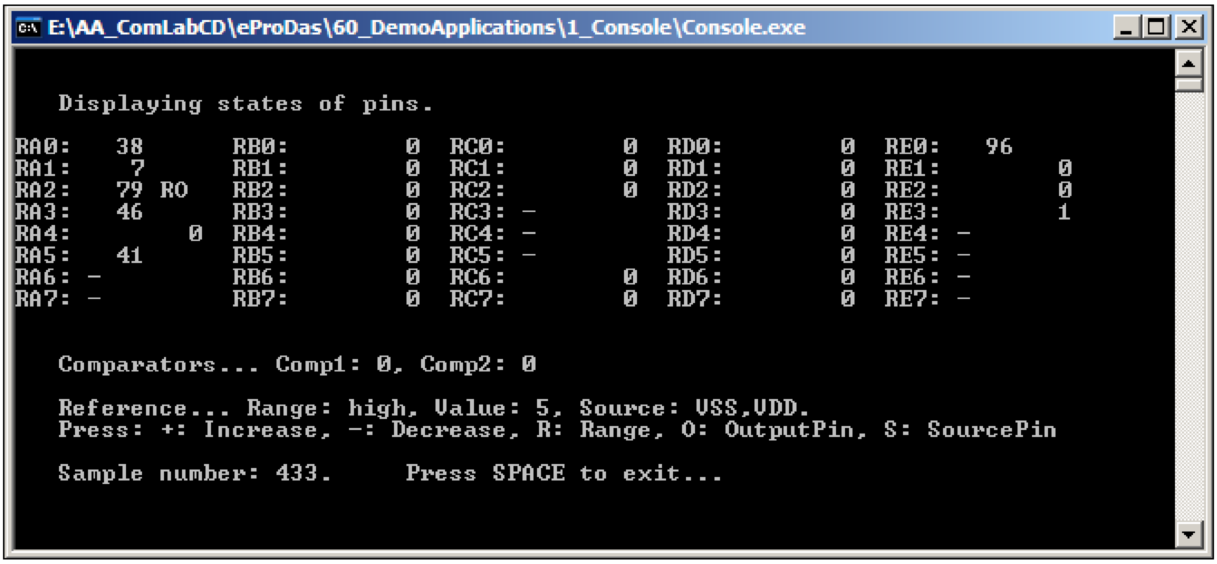
```
Reference module can produce voltages in two different ranges.

Low range:  voltages from 0 V to 0.667*Vref in steps Vref/24.
            Vout = (Vref+ - Vref-)*(Value/24).

High range: voltages from 0.25*Vref to 0.75*Vref in steps Vref/32.
            Vout = Vref- + (Vref+ - Vref-)*0.25 + (Vref+ - Vref-)*(Value/32).

Enter 1 for high range and 0 for low range:  (from 0 to 1, -1 to cancel): 1
Enter Value:  (from 0 to 15, -1 to cancel): 5

-----   Command completed successfully -----




----- Device Information ----------------------------------------------------

TYPE: Device Type: 1 (Single Chip), Device Tag: 255
INFO: Firmware: 0.4, DLL: 0.4, KDrv: 0.4
DIAG: USB Addres: 2

----- Device Configuration --------------------------------------------------

RA0: AD,C-     RB0: DI        RC0: DI        RD0: DO        RE0: AD
RA1: AD,C-     RB1: DI        RC1: DI        RD1: DO        RE1: DI
RA2: AD,RO     RB2: DI        RC2: DI        RD2: DO        RE2: DI
RA3: AD,C+     RB3: DI        RC3: -         RD3: DO        RE3: DI
RA4: DI        RB4: DI        RC4: *         RD4: DO        RE4: -
RA5: AD        RB5: DI        RC5: *         RD5: DO        RE5: -
RA6: *         RB6: DI        RC6: DI        RD6: DO        RE6: -
RA7: -         RB7: DI        RC7: DI        RD7: DO        RE7: -

AD Module: Channel: 0/6, VRef-: Vss, Vref+: Vdd, Clock: 750000 Hz
Conversion: 14.7 mus (11 clocks), Acquisition time: 0 mus (0 clocks)


Comparator module: mode/switch: 5/0, InvertC1: 0, InvertC2: 1.

Reference module: OutputOnRA2: 1, SourceOnPins: 0, Range: high, Value: 5.

===== MAIN MENU =============================================================

<: OpenHandle, >: CloseHandle,          ?: PrintRegisters, !: Refresh, @: End
\: SysInit,    |: ClearDiag,   Y: PinStates

1..5: Write Ports A..E, 6..0: Configure Ports A..E
A: Configure AD, C: Configure Comparators, R: Configure Reference

Config O: PWM1, P: PWM2; DutyCycle U: PWM1, I: PWM2; Off K: PWM1, L: PWM2
H: Change PWM1 Direction (full-bridge mode), J: Clear PWM1 Auto Shutdown

M: Periodic actions

=============================================================================
```

**Figure 33: An example of a more elaborate device configuration.**

By examining the description, we can realize that AD module is turned on and 6 out of 13 possible AD inputs are configured for analogue sampling. These are pins from RA0 to RA3, RA5 and RE0. Pins RA0 and RA1 are also inverting inputs of the associated voltage comparators. Pin RA2 also outputs the reference voltage and pin RA3 is a non-inverting input of the associated voltage comparator. In addition, the whole PORTD, which consists of pins RD0 through RD7, is configured as digital output. All these features are easily configured by executing commands in the `Main Menu`.

Please note, that the presented features are not orthogonal or independent of each other. The PIC18F4550 does not allow e.g. to configure pins RA0 and RE0 as inputs to AD converter without also configuring pins RA1, RA2, RA3 and RA5 in this way. Further, inputs and output of comparators and voltage reference cannot be moved to other pins. These features are described in detail later.

When the system is configured according to your preferences, you can monitor states of pins in real time by selecting command `PinStates`. The example screen is presented in the Figure 34.


E:\AA_ComLabCD\eProDas\60_DemoApplications\1_Console\Console.exe

```

   Displaying states of pins.

RA0:   38      RB0:        0  RC0:        0  RD0:        0  RE0:   96
RA1:    7      RB1:        0  RC1:        0  RD1:        0  RE1:        0
RA2:   79 RO   RB2:        0  RC2:        0  RD2:        0  RE2:        0
RA3:   46      RB3:        0  RC3: -         RD3:        0  RE3:        1
RA4:        0  RB4:        0  RC4: -         RD4:        0  RE4: -
RA5:   41      RB5:        0  RC5: -         RD5:        0  RE5: -
RA6: -         RB6:        0  RC6:        0  RD6:        0  RE6: -
RA7: -         RB7:        0  RC7:        0  RD7:        0  RE7: -


   Comparators... Comp1: 0, Comp2: 0

   Reference... Range: high, Value: 5, Source: VSS,VDD.
   Press: +: Increase, -: Decrease, R: Range, O: OutputPin, S: SourcePin

   Sample number: 433.     Press SPACE to exit...
```

**Figure 34: Monitoring state of pins.**

Pins RA0 through RA3 and RA5 as well as RE0 are inputs of AD converter, so they display “analogue” value on their pins, which is left aligned in the respective column. Pin RA2 also outputs the voltage of the reference voltage module (hence the mark `RO`).

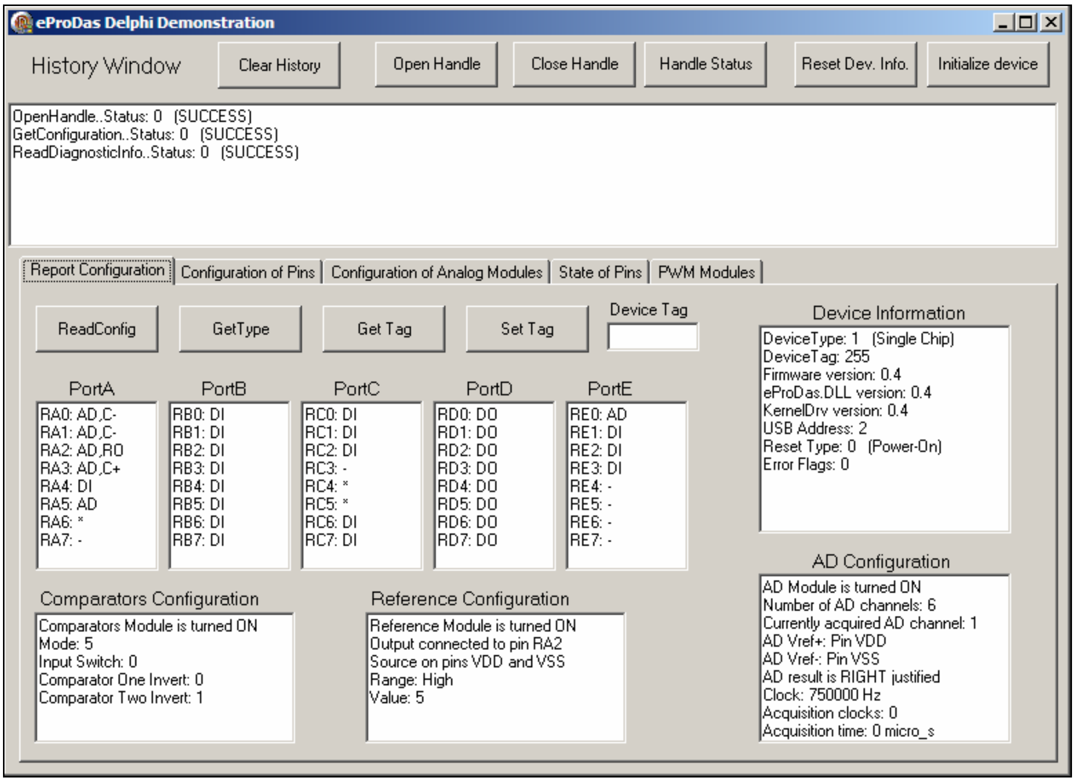
Other pins are digital inputs or outputs. In the case of a digital pin its state is displayed right aligned. We can see that only pin RE3 is in state logic 1, since in our case this pin is connected by the pull-up resistor to VDD supply rail, which we need for proper working of In-Circuit debugger.

If you change states on the pins by external circuitry, the modified values will be immediately reflected on the screen. You can also monitor the outputs of the voltage comparators even if they are not connected to external pins. In addition, you may change the value of voltage reference by pressing the appropriate keys as it is instructed at the bottom of the console screen and hook up a Voltmeter to pin RA2, which will reflect the voltage changes on the pin.

# 9 Demo application in Delphi

There exists another demo application that is written in the Delphi development environment. You can find it in the folder `eProDas/40_DemoApplications/2_Delphi`. Contrary to the previously described console application this one has got a graphical user interface, which you should find more convenient and user friendly. However, the primary role of the application is to be a diagnostic tool, by means of which we test the proper working of the eProDas Delphi library. Therefore, the user interface of the application is not polished for the ultimate experience nor does this piece of code exposes capabilities of eProDas devices in a much useful way.

Figure 35 presents the first screen of the application, where you can see that this program again tries to summarize a lot of device information in a pure textual form. The roles of the PIC's pins are annotated in the same way as in the case of the console application. Device information and configuration of other modules are also reported to you.



```
OpenHandle..Status: 0  (SUCCESS)
GetConfiguration..Status: 0  (SUCCESS)
ReadDiagnosticInfo..Status: 0  (SUCCESS)
```

Device Information

```
DeviceType: 1   (Single Chip)
DeviceTag: 255
Firmware version: 0.4
eProDas.DLL version: 0.4
KernelDrv version: 0.4
USB Address: 2
Reset Type: 0   (Power-On)
Error Flags: 0
```

| PortA | PortB | PortC | PortD | PortE |
|---|---|---|---|---|
| RA0: AD,C- | RB0: DI | RC0: DI | RD0: DO | RE0: AD |
| RA1: AD,C- | RB1: DI | RC1: DI | RD1: DO | RE1: DI |
| RA2: AD,RO | RB2: DI | RC2: DI | RD2: DO | RE2: DI |
| RA3: AD,C+ | RB3: DI | RC3: - | RD3: DO | RE3: DI |
| RA4: DI | RB4: DI | RC4: * | RD4: DO | RE4: - |
| RA5: AD | RB5: DI | RC5: * | RD5: DO | RE5: - |
| RA6: * | RB6: DI | RC6: DI | RD6: DO | RE6: - |
| RA7: - | RB7: DI | RC7: DI | RD7: DO | RE7: - |

Comparators Configuration

```
Comparators Module is turned ON
Mode: 5
Input Switch: 0
Comparator One Invert: 0
Comparator Two Invert: 1
```

Reference Configuration

```
Reference Module is turned ON
Output connected to pin RA2
Source on pins VDD and VSS
Range: High
Value: 5
```

AD Configuration

```
AD Module is turned ON
Number of AD channels: 6
Currently acquired AD channel: 1
AD Vref+: Pin VDD
AD Vref-: Pin VSS
AD result is RIGHT justified
Clock: 750000 Hz
Acquisition clocks: 0
Acquisition time: 0 micro_s
```

**Figure 35: The first screen of the Delphi demo application.**

**Note.** The Figure 35 may not reflect the exact contents of your screen since the screenshot may be outdated.

Configuration of various modules and implemented observation of pin states may be found under the proper tabs that are visible in the figure.

**Note.** This demo application does not support multiple connected eProDas devices (see the next chapter).

# 10 Using various and multiple devices in your setup

Since the eProDas platform (section 3.1) does not denote one particular device but rather a whole family of devices, sooner or later someone will come into situation where she or he utilizes one type of device for certain experiment and another type of device for some completely unrelated work. In such cases it is useful for an end-user application to be able to check, whether the proper one of the two devices is connected to a host PC. Generally, there should exist some sort of hardware type identification, which end-user application can query to learn about the actual device that is connected to a host PC at a moment. Entering Device Type.

## 10.1 Device Type

Device Type is nothing more than a plain 8-bit number, which the developer or the producer of the device (this could by you) assigns to her or his masterpiece. The chosen Device Type is stored in PIC's data EEPROM and is therefore a non-volatile parameter of the device.

The eProDas team reserves the first 64 Device Type numbers (from 0 to 63) for devices that we intend to develop, whereas you are free to consume the rest of the choices at your will. The number 255 is also reserved and it denotes the unspecified Device Type; the selection is dictated by the physiology of EEPROM memory, which contains all ones when erased.

The next table lists the already defined Device Types and briefly describes the hardware that they are associated with.

| Acronym | Abbreviation | Device Type | Description |
|---|---|---|---|
| Unspecified | / | 255 | Indicator that Device Type has not been assigned (yet). |
| Platform | PL | 0 | Generic eProDas platform and the basis for development of other members of the family. |
| SingleChip1 | SC1 | 1 | PCB with implementation of schematic in the Figure 3 with added connectors for each PIC's pin. |
| SingleChip2 | SC2 | 2 | PCB with implementation of schematic in the Figure 3, except that PIC18F4550 is replaced with PIC18F2550. There are also connector for SPI bus, two connectors for ultra sound sensors and connectors for a few general purpose PIC's pins. |
| MultiFunction1 | MF1 | 3 | Not entirely specified yet… it will contain a few external AD converters (probably 12-bit, with SPI bus) and possibly other capabilities. |

**Table 1: Official members of the eProDas family and their Device Types.**

To assign (change) the Device Type to (of) your device, run the SetType utility, which you can find in the folder 70_Utilities/1_SetTag of the eProDas installation disk. This utility is self explanatory and if you follow the instructions on screen, you should have no trouble successfully assigning Device Type designator to a device. During the process make sure that only one eProDas device is connected to USB bus, since the SetType utility cannot know to which one of them you want to assign a particular Device Type.

**Note.** There also exists API function (section 23.1.1) for assigning device type designators within your applications.

## 10.2 Working with multiple devices simultaneously

eProDas system can work with more than one eProDas device connected to a host PC (chapter 21), which makes it more adoptable to various situations that may arise in practice. As an example, let us speculate that fifty percent of natural science experiments demand two AD channels. To cover additional twenty five percent we need three AD channels. For ninety percent coverage of the possible situations four of them are needed, etc.

There are two ways to give the needed AD channels to the people. The first option is to squeeze a large number of them on a system by means of which we cover all but the most demanding cases. This brute forte approach has many drawbacks. Namely, all users would pay a bigger price for the system, since more parts and features inevitably lead to increase of costs. The system would be physically larger and therefore less convenient to use and it would also consume more power by which an external power supply would be necessary in many cases, where we could avoid it otherwise, or perhaps the batteries of your laptop would be brought to their knees sooner.

A better solution is to make the system more flexible and adoptable by allowing it to work with more than one eProDas device at the time. If, for example, your device has got two AD channels but you need three of them, hook the second device into an empty USB slot and there you are. According to this doctrine the users, which do not need the excessive number of AD channels, are not forced to buy them nor need they to feed them with the power.

When working with multiple eProDas devices your application needs a way to distinguish them according to their assigned roles. For example, you are measuring characteristics of a black box electronic circuit. One eProDas device is devoted to measuring the instant voltages and currents at the input of the circuit and similarly another one is dedicated to observing the output responses. In order for the experiment to be meaningful your application needs to know which eProDas device measures what quantities.

The question is how to do the differentiation, especially when you utilize two eProDas devices of the same type, i.e. with equal `Device Type` designator. A no-plausible solution is to use their USB addresses that Windows assigns to them and which are guaranteed to be unique. Although possible (section 24.1.1), we strongly discourage you to do it.

Namely, the USB standard does not specify the enumeration process by which USB addresses are assigned to the devices. Although the current Windows enumeration algorithm appears to be deterministic and repeatable there is no guarantee that it will not change in the future versions of Windows or even when a new patch or service pack is applied to the current version of Windows.

Further, if you unplug the devices from the USB and re-plug those back in, you have to use exactly the same USB slot arrangements for the eProDas as well as for all other USB devices in order to preserve their USB addresses. This is a frustrating limitation especially if you need to plug your eProDas constellation into another computer to make a public presentation. There has to be a better way…

However, it seems that the issue was resolved in Windows Vista, which handles eProDas devices in a more intelligent way.

## 10.3 Device tags

The described issue is solved by utilization of the so called device tags, which are (again) non-volatile 8-bit numbers that are stored into the PIC's on chip EEPROM. When multiple eProDas devices are used for the experiment, you simply tag each one of them with a unique value of your choice, so that your application can do its own and USB independent enumeration of the devices.

To tag the device use the `SetTag` utility, that is located in the folder `70_Utilities/1_SetTag` of the eProDas installation disk. This utility is self explanatory and by following the instructions on screen, you should have no trouble successfully tagging the device. During the process make sure that only one eProDas device is connected to the USB bus, since the `SetTag` utility cannot differentiate eProDas devices before they are tagged and consequently it cannot know to which one of them you want to assign a particular tag.

**Note 1.** If device has not been tagged yet, its tag value equals 255, which is due to the fact that erased EEPROM contains all bits set to one.

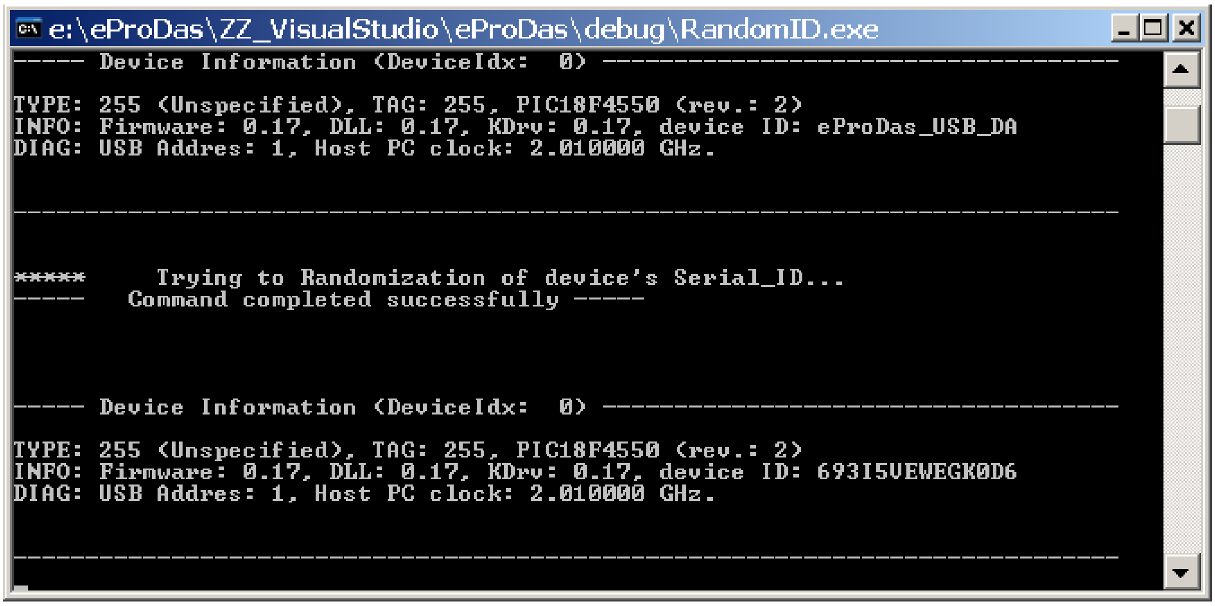
**Note 2.** There also exists API function (section 23.1.2) for tagging devices within your applications.

## 10.4 Device IDs and accompanying awkwardness

To be able to work with several equal devices simultaneously eProDas system must comply with Windows requirement that each connected device has a unique identifier (Serial ID). In the case of eProDas system such uniqueness is achieved by writing a random Serial ID to the device's program RAM, which then device reports to the Windows whenever it is plugged into the USB connector or when the system reboots.

The eProDas Serial ID is a fourteen character string, where each character can hold a number from 0 to 9 or capital letter from A to Z. Since each character holds one of 36 possible values and there are 14 of such values there can exist more than $6 \cdot 10^{21}$ unique IDs; enough that the odds of two devices at your desk having the same ID is negligibly small.

When eProDas device is programmed or reprogrammed by using Microchip's ICD-2 or some other piece of equipment, the Serial ID is set to a generic value of "eProDas_USB_DA". To replace this pre-assigned ID with a random one, run a utility `RandID`, which is located in the folder `eProDas\70_Utilities\2_UpgradeFirmware` of the ComLab installation disk. The screenshot of the utility in action is presented in the following figure.



```
----- Device Information (DeviceIdx:  0) ------------------------------------

TYPE: 255 (Unspecified), TAG: 255, PIC18F4550 (rev.: 2)
INFO: Firmware: 0.17, DLL: 0.17, KDrv: 0.17, device ID: eProDas_USB_DA
DIAG: USB Addres: 1, Host PC clock: 2.010000 GHz.


-----------------------------------------------------------------------------


*****     Trying to Randomization of device's Serial_ID...
-----    Command completed successfully -----



----- Device Information (DeviceIdx:  0) ------------------------------------

TYPE: 255 (Unspecified), TAG: 255, PIC18F4550 (rev.: 2)
INFO: Firmware: 0.17, DLL: 0.17, KDrv: 0.17, device ID: 693I5VEWEGK0D6
DIAG: USB Addres: 1, Host PC clock: 2.010000 GHz.


-----------------------------------------------------------------------------
```

**Figure 36: Screenshot of RandID utility.**

In the first part of the screen the information about the attached device is displayed. At the end of the row “INFO” the current device’s ID is listed. Since our device has just been baked it has a generic ID “eProDas_USB_DA”.

In the second part of the screen the utility informs us about its effort to randomize the ID. The success or failure of the operation is also reported.

The last part of the screen presents the updated information about the device, where a newly assigned device ID can be spotted (if you are curious about its value).

**Note.** Until the device is unplugged and re-plugged back (or the system is rebooted) the new ID is not reported to the Windows.

### 10.4.1 Boring and unnecessary reinstallations of device driver

After you successfully randomize your device unplug it from the USB and re-plug it back in. Now we are going to discover the awful truth. Windows insists on installing the device driver again and you have no choice but to comply. Worse still, now after the driver is installed for the second time, unplug the device and re-plug it into some other USB port to discover that the driver needs to be re-installed again. Yes, the same driver each time with the same files `eProDas.sys` and `eProDas.DLL`. Enough trouble to drive you crazy.

Fortunately, it seems that at a certain point (when those installations have bitten you to the dust) Windows stops being so stupid and you can connect device with different Serial ID without an unnecessary reinstallation hard work. Exactly when or why does this happen is still a mystery to us and according to relevant Internet forums the same problem seems to trouble other devices too. If you have any clue whatsoever, please reveal us this mighty secret.

# 11 Upgrading your eProDas system

From time to time we might release updates of the various parts of the eProDas device stack. Generally, you will want to apply these updates to your system, since they will expand the functionality of eProDas devices or correct the discovered bugs.

**Note.** You must take great care to upgrade all parts of the eProDas stack with the jointly released pieces of it. Namely, if you only upgrade, say, firmware but you forget to upgrade device driver your system may cease to work. It is even possible that the majority of functions do work and you think that your system is okay, but some infrequently used feature may be broken.

Generally, upgrade process should be done according to guidelines in the following sections. However, it is possible that due to a certain bug in the previous (i.e. your current) release of device stack you need to perform special steps to successfully complete the job. In such situations the updated software is accompanied with detailed instructions about the actual upgrade process.

**Always read these materials in their entirety before proceeding with the upgrade.**

## 11.1 Upgrading the firmware which should be ALWAYS upgraded first

In order to program new firmware in the PIC you needed a PIC programmer or person with it at hand. However, if you already had to ask a friend or someone less friendly to program your chip because you do not have the programming device, it would be a nuisance to go through the process again and again for applying each released patch. To make the things as convenient for you as possible, we developed a method to upgrade you old firmware with a new one without the PIC programmer.

All you need to possess is a HEX file (section 6.1) with an updated version of firmware and a small utility `UgradeFirmware`. You can find the latter on the eProDas installation disk under the directory `50_Utilities/2_UpgradeFirmware`. Whenever you want to freshen up your current version of firmware with the new one, simply fire up this utility and follow the instructions on screen.

Before running the `UpgradeFirmware` utility, please note the following.

**If not instructed otherwise by documentation that accompanies the new version of firmware, you should ALWAYS perform firmware upgrade by using OLD elements of the device's stack. Do NOT use Windows device driver, `eProDas.DLL` and `UpgradeFirmware` on a new installation media. Instead, simply copy new version of file `eProDas_M_m.HEX` into directory with the old `UpgradeFirmware`, which in addition still uses the old version of `eProDas.DLL` and Windows device driver. Only after firmware is successfully upgraded, proceed with upgrading other pieces of eProDas device stack.**

**If the firmware upgrade process fails it might happen that your eProDas device will not be operational any more. In this case, you will have to visit your dear friend with the PIC programmer one more time to restore the functionality of the device.**

We did our best to minimize the risk of failed upgrades. In almost all erroneous cases the device should be able to either continue using the old firmware, which gives you the possibility to repeat the upgrade, or succeed in finishing the upgrade. Despite these claims please have in mind that erroneous cases are diverse by their nature and that exhaustive testing of the upgrade process by many users of this feature is necessary to evaluate its robustness in a meaningful way. Currently no such large scale testing has been done yet so the best you and we can do is to have our fingers crossed.

### 11.1.1 Firmware upgrade and device tags

For your convenience the utility `UpgradeFirmware` preserves `Device Type` and `Device Tag` of the upgraded device. However, if you do the upgrade by reprogramming the device with the Microchip In-Circuit debugger or some other hardware programmer, these two settings may not be preserved depending on the configuration and capabilities of your programming device. In order to preserve the settings, you have to make sure that your programmer does not erase data EEPROM of the device. Follow the instructions for your particular programming device to learn how to achieve this functionality. Alternatively, you can restore the device's setup by rerunning the `SetType` (section 10.1) and/or `SetTag` (section 10.3) utilities.

### 11.1.2 Firmware upgrade and Serial ID

The utility UpgradeFirmware also preserves the previous Serial ID if it is already randomized. However, if the current (i.e. old) Serial ID equals the generic "eProDas_USB_DA" serial ID or if any character of the Serial ID string is invalid, the new randomized ID is written to the device during the upgrade process.

## 11.2 Upgrading the device driver and eProDas.DLL

To upgrade the device driver and `eProDas.DLL` follow the instructions in section 0 to arrive at the dialog box in the Figure 25 (right). Click on the `Update driver` button (which will be disabled if you do not have administrative privileges as it is the case in the figure) and perform the upgrade as it would be a fresh driver installation according to the instructions in section 7.

# 12 Supported operating systems

So far the eProDas system is developed and run on the Microsoft Windows XP Professional operating system. According to our testing Windows Vista is okay as well. eProDas should also work on all incarnations of Windows 2003 Server, but we did not test it. As far as we can tell, it should also work on Windows XP Home Edition, at least according to the press information about the same kernel base. If you test the eProDas on any of these systems, please inform us about the outcome to clear and clarify the usable set of operating systems. Also, Windows 2000 might be fine for the job, however this operating system is outdated and we do not intend to support it or hunt bugs that seem to appear only on this platform. The eProDas system will NOT work on Windows 95, 98 and Millennium.

We are interested in porting the system to Linux platform, but we do not have the necessary expertise. If you are willing to help us with this issue or you are at least kind enough to tell us the shortest possible learning path then the Linux port may see the sun some day.

# 13 Known Issues

1. In order to fully comply with the USB rules it would be necessary to register our vendor ID to the USB consortium and pay the registration fee (currently 1500 $). Since, this has not been done, we do not have an official vendor ID and we use the made up one, which currently equals 0xF0F0. If a registered vendor with such ID truly exists and you have any of its devices installed on your system then Windows will be confused and possibly both, the eProDas and that device, will not work properly.

   In the case of vendor ID clash, the workaround is to change the vendor ID and/or the product ID in the eProDas firmware as well as adjust the file `eProDas.INF` and to reinstall the eProDas device driver. These steps need certain expertise so they are not meant to be done by an average eProDas user. If you encounter the described problem, contact us and we will try to solve it.

2. The device driver `eProDas.sys` is currently a half baked cake, which only supports basic functionality of eProDas device, but no additional services that are needed for proper working of your operating system. The most annoying issue is the complete absence of the power management functionality, with the consequence that Windows cannot switch to any low power modes, such as standby or hibernation, when the driver is operational.

   As a workaround close all programs that access the eProDas functionality and unplug the eProDas device(s) from the USB cable. This will unload the driver from Windows memory and stop inhibiting your system from entering the low power state of your choice.

# Programming manual

# 14 Overview

End user applications access the functionality of eProDas devices through a set of application programming interface (API) functions that are implemented in the library `eProDas.DLL`. Function prototypes obey declaration and linkage specifications of the ANSI C programming language, although the `eProDas.DLL` is written in the C++ programming language. In order to assure smooth integration of eProDas API functions with other programming languages and development tools, we tried to minimize the use of structures and classes for function parameters. Whenever possible or aesthetically permissible the pure scalar parameters were preferred.

In addition to the C programming language `eProDas.DLL` library we prepared libraries for Delphi, Visual Basic (not completed) and LabView (not done yet) developers, to assure that the eProDas system integrates natively with these development environments. Please, read the appropriate one(s) of the following paragraphs on how to gain the access to eProDas functions in your programming language and development environment.

### 14.1.1 The C/C++ programming language

For the C/C++ developers we prepared a file `eProDas.h`, by means of which your compiler learns everything it needs to know about the eProDas functions. This file is located in the folder `eProDas/50_API/1_CPP_Libraries` of the eProDas installation disk. You should `#include` the file `eProDas.h` into every C or CPP file that communicates with eProDas devices.

In addition, you must make sure that the library `eProDas.DLL` becomes part of your project. If you are developing your eProDas applications with Microsoft's Visual Studio integrated development environment you can do it easily by statically linking the file `eProDas.lib` in the above mentioned folder with your application. This file takes care of loading and calling functions in `eProDas.DLL` behind the scenes.

In the case that your particular development environment cannot link your code with the file `eProDas.lib`, you have to take the burden of loading the `eProDas.DLL` library to your shoulders. The procedure is the same as for any other `DLL` library. Follow the instructions for your specific compiler.

### 14.1.2 The Delphi development environment

To work with eProDas devices in Delphi, add the unit `eProDas.pas` to the `Uses` clause of the appropriate module; the unit is located in the folder `eProDas/50_API/2_DelphiLibraries`. Nothing else is needed, since Delphi automatically takes care of loading the library `eProDas.DLL`.

### 14.1.3 The Visual Basic development environment

To work with eProDas devices in Visual Basic, add module `eProDasModule.vb` to your project. This file is located in the folder `eProDas/50_API/3_VisualBasicLibraries` of the ComLab installation disk. Nothing else is needed, since loading of the library `eProDas.DLL` is automatic.

Note. Some eProDas functions are not supported in this development environment due to limited support for manipulation of pointers.

## 14.2 eProDas error codes

Neglecting few exceptions, each eProDas API function returns error code as a signed integer result. This integer indicates the success or the failure of the operation that the function is supposed to perform. The possible error codes are listed in the Table 2 and Table 3 on the next pages.

As the name suggests, the code `eProDas_Error_SUCCESS` is the only error code that indicates a successful operation. If function returns any other error code, the requested operation is not completed as expected and the reason for failure is indicated by the returned value.

**Note.** In the case of error, eProDas functions do not try to undo partial changes, which may be applied to the eProDas system prior to reaching the erroneous condition. Functions can neither be considered atomic operations nor do they mimic transactions like semantics. Generally, there is no need to do it and implementing such behaviour would impose certain overhead, which may become a performance bottleneck. If your application is written well, there should be no error conditions at all, except if something is wrong with your device (like accidental unplugging of the USB cable). Therefore, these error codes could (and should) be perceived as a debugging tool as well as hardware diagnostics mechanism.

The libraries for the supported programming languages define appropriate constants for all error codes, listed in the Table 2. We strongly suggest you to use constants instead of pure numbers in your programs since the actual values of the codes may change in the future. Only the value of the code `eProDas_Error_SUCCESS` is guaranteed to remain 0, since such choice leads to code simplification or speedup when you are only interested whether the function executed correctly or not, but you do not care about the particular reason for failure.

**You should always check error code after each eProDas function call and abort the operation or recover gracefully in the case of failure.**

Walking along with blind eyes may work for a while but in the unfortunate and unpredictable cases the time spent on seeking the failure without proper error-checking infrastructure becomes a tedious and boring task. Also, your application may work completely flawlessly until someone accidentally unplugs USB cable from the host PC. If the application fails to observe such erroneous state, it will continue working at unchanged pace and put a shadow of shame on its developer at some public presentation. Not to mention even more likely scenario of doing some flawed measurements without knowing about the failure and publishing the results in some scientific journal.

| Name | Value |
|---|---|
| eProDas_Error_SUCCESS | 0 |
| eProDas_Error_FailedToOpenHandle | –1 |
| eProDas_Error_HandleAlreadyOpen | –2 |
| eProDas_Error_FailedToCloseHandle | –3 |
| eProDas_Error_InvalidDeviceIdx | –4 |
| eProDas_Error_WriteToUSBFailed | –5 |
| eProDas_Error_ReadFromUSBFailed | –6 |
| eProDas_Error_InternalError | –7 |
| eProDas_Error_UnimplementedFunctionality | –8 |
| eProDas_Error_InvalidMemoryBlock | –9 |
| eProDas_Error_WriteToDataEEPROMFailed | –10 |
| eProDas_Error_WriteToProgramFLASHFailed | –11 |
| eProDas_Error_InvalidDeviceTag | –12 |
| eProDas_Error_UpgradeFirmwarePhaseOneFailed | –13 |
| eProDas_Error_UpgradeFirmwarePhaseTwoFailed | –14 |
| eProDas_Error_OpenFileFailed | –15 |
| eProDas_Error_ReadFromFileFailed | –16 |
| eProDas_Error_InvalidFileFormat | –17 |
| eProDas_Error_FirmwareTooBig | –18 |
| eProDas_Error_InvalidPortIndex | –19 |
| eProDas_Error_InvalidNumberOfADChannels | –20 |
| eProDas_Error_InvalidAcquisitionTimeCode | –21 |
| eProDas_Error_InvalidADChannel | –22 |
| eProDas_Error_InvalidComparatorMode | –23 |
| eProDas_Error_InvalidReferenceValue | –24 |
| eProDas_Error_InvalidMode | –25 |
| eProDas_Error_InvalidPeriod | –26 |
| eProDas_Error_InvalidPrescaler | –27 |
| eProDas_Error_InvalidDutyCycle | –28 |
| eProDas_Error_InvalidActiveState | –29 |
| eProDas_Error_InvalidDelay | –30 |
| eProDas_Error_InvalidAutoShutDownSource | –31 |
| eProDas_Error_InvalidAutoShutDownPinState | –32 |
| eProDas_Error_TimerDisabled | –33 |
| eProDas_Error_BadMainDivider | –34 |
| eProDas_Error_BadExtendedDivider | –35 |
| eProDas_Error_ClockTooHigh | –36 |
| eProDas_Error_ClockTooLow | –37 |
| eProDas_Error_ClockNotPossible | –38 |
| eProDas_Error_ResourceInUse | –39 |
| eProDas_Error_PeriodicActionsNotActive | –40 |
| eProDas_Error_NotEnoughSpace | –41 |
| eProDas_Error_InvalidParameter | –42 |
| eProDas_Error_PreviousValueOverwritten | –43 |
| eProDas_Error_InvalidChainSize | –44 |
| eProDas_Error_InvalidPeriodicProgram | –45 |
| eProDas_Error_IN_PacketSizeNotMultipleOfOnePeriod | –46 |
| eProDas_Error_OUT_PacketSizeNotMultipleOfOnePeriod | –47 |
| eProDas_Error_WriteToCONFIGByteFailed | –48 |
| eProDas_Error_InvalidPeriodicConfiguration | –49 |
| eProDas_Error_ModuleNotActive | –50 |

**Table 2: eProDas error codes (part one).**

| Name | Value |
|---|---|
| eProDas_Error_Invalid_IN_PeriodicPacketSize | –51 |
| eProDas_Error_Invalid_OUT_PeriodicPacketSize | –52 |
| eProDas_Error_PeriodicProgramTooLong | –53 |
| eProDas_Error_InvalidFileRegister | –54 |
| eProDas_Error_InvalidBitNumber | –55 |
| eProDas_Error_InvalidDisplacement | –56 |
| eProDas_Error_InvalidAddress | –57 |
| eProDas_Error_InvalidLiteral | –58 |
| eProDas_Error_InvalidFSR | –59 |
| eProDas_Error_InvalidPinIndex | –60 |
| eProDas_Error_DuplicateLabel | –61 |
| eProDas_Error_InvalidInstruction | –62 |
| eProDas_Error_DataCorruptionError | –63 |
| eProDas_Error_NotEnoughResources | –64 |
| eProDas_Error_EndOfBufferReached | –65 |
| eProDas_Error_InvalidDuringPeriodicActions | –66 |

**Table 3: eProDas error codes (part two).**

### 14.2.1 Description of common errors

Some of the errors in the Table 2 and Table 3 may arise during execution of any function and can be viewed as general errors, still others are function specific ones. The former are described in this section, in order not to repeat their descriptions for each function that is discussed further on.

1. `eProDas_Error_WriteToUSBFailed` is returned whenever `eProDas.DLL` tries to send some data to the eProDas device, which the latter refuses to accept. This may be due to the following reasons.

   - The eProDas device handle is not open (see section 14.3).
   - The eProDas device was unplugged from the cable.
   - The hardware is not operational, e.g. a wire on your breadboard was accidentally disconnected.
   - Any piece of the eProDas device stack has a bug.
   - If you are using a developer set of functions you might destroy the functionality of the device by erasing or reprogramming the vital parts of the firmware or its variables.

2. `eProDas_Error_ReadFromUSBFailed` is similar to the previous error, except that the `eProDas.DLL` tries to receive some data from eProDas device, which the latter refuses to send. The possible causes are completely the same as in the previous case. Two distinct errors exist merely to assist us (developers of eProDas device stack) in hunting bugs by reporting erroneous conditions with a finer granularity than in the case of a unique error code.

3. `eProDas_Error_InternalError` this error indicates a bug in the `eProDas.DLL`, Windows device driver or firmware. The exchange of data between the components of device stack is working well (i.e. your hardware is probably ok), but the actual contents are wrong. Also device or code execution may have reached invalid state, which is again indication of a bug in some piece of device stack and is therefore ours and not yours responsibility.

4. `eProDas_Error_InvalidDuringPeriodicActions` indicates that your application is trying to execute *ordinary* commands during active periodic actions (chapter 19), which cannot be done due to the internal workings of periodic actions.

Please, note. The first four errors (especially `eProDas_Error_InternalError`) may be due to the inconsistencies between the different versions of the pieces of software that form the eProDas device stack. For example, if you upgrade firmware with a renewed one, but you continue to use the old version of `eProDas.DLL`, they may not understand each other any more because of the changes in their internals. Always make sure that you upgrade all parts of your eProDas system according to the instructions in chapter 11.

## 14.3 eProDas device handle

From the `eProDas.DLL` point of view your eProDas device behaves almost the same as a file on a disk. Accordingly, `eProDas.DLL` communicates with the device through Windows system calls that operate on files, such as file open (`fopen` in C/C++), file write… The role of `eProDas.sys` device driver is to represent physical eProDas device with its peculiarities to `eProDas.DLL` as a file.

The library `eProDas.DLL` knows how to read from and write to this eProDas file to perform AD conversions, configure pins and do all other tasks that are described herein after. The complexity of this actual low-level file communication is entirely encapsulated into the library `eProDas.DLL` and is completely hidden to you, the application developer.

Still, before the `eProDas.DLL` can perform its job it has to open a handle to the eProDas device (i.e. file) as it is required for any file operation. Exactly, when it is the right moment to do it is under your control and for that matter there exist two eProDas API functions especially for working with device handles.

**Note.** You must open device handle before you can use any other eProDas functions.

### 14.3.1 Function OpenHandle

Prototype in C/C++

```
int eProDas_OpenHandle();
```

Prototype in Delphi

```
function eProDas_OpenHandle:integer;
```

Prototype in VisualBasic

```
Function eProDas_OpenHandle() As Integer
```

This function tries to open eProDas device handle, which will succeed if at least one operational eProDas device is attached to the USB bus and all parts of the eProDas device stack are installed correctly. The generally suitable time for calling this function is at the beginning of your application or whenever before performing the first action on the eProDas device.

In the case of the failure the error code `eProDas_Error_FailedToOpenHandle` is returned. The most probable cause is that the eProDas device is not attached to the USB bus or it is not operational. This function also fails if the eProDas Windows device driver is not installed correctly.

The third possibility is that the user clicked on the option “Safely remove the eProDas data acquisition system” in the Windows system tray, which removed the device from the Windows system. As the Windows is concerned the device is detached from the USB bus, although it is still physically present. To remedy the situation truly disconnect the device from the USB and close all applications that may be trying to access the device. A few moments later plug the device back in and try again.

**Note.** If you do not succeed in opening the handle, you cannot access any functionality of the device.

### 14.3.2 Function CloseHandle

Prototype in C/C++

```
void eProDas_CloseHandle();
```

Prototype in Delphi

```
procedure eProDas_CloseHandle;
```

Prototype in VisualBasic

```
Sub eProDas_CloseHandle()
```

With this function you close the previously opened eProDas device handle. Usually, you do this at the end of your application, but you may decide to do it at any time, when you know that your application does not need the functionality of the device any more. For example, if you have finished with measurements but you have to analyze the results and do some calculations for a couple of more hours, you may release device handle even if your application is far away from finishing its job so that some other application may access the device during this idle device time.

**Note.** If you fail to close the handle the driver `eProDas.sys` will not allow any other eProDas application to open the handle as long as your application is running.

## 14.4 Device indexes

eProDas system supports simultaneous operation of more than one eProDas devices (chapters 10 and 21). For that matter, the first parameter of the majority of eProDas functions is an unsigned integer named `DeviceIdx`. With it the application informs API function about the device that it wants to influence. The exact semantics of the parameter is described in chapter 21. When only one device is connected to the host PC set the parameter `DeviceIdx` to 0.

When specifying a non-existing device by setting `DeviceIdx` to an invalid value, eProDas functions return with error code `eProDas_Error_InvalidDeviceIdx`.

## 14.5 Device initialization

Whenever you work with true hardware you often need a function that puts the device into the initial predefined state. In the case of eProDas the function for the task is `Initialize`.

### 14.5.1 Initialize

Prototype in C/C++

```
int eProDas_Initialize(unsigned int DeviceIdx);
```

Prototype in Delphi

```
function eProDas_Initialize(DeviceIdx: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_Initialize(ByVal DeviceIdx As UInteger) As Integer
```

The function `OpenHandle` automatically initializes all discovered (enumerated) eProDas devices, so you do not need to call the function `Initialize` explicitly at the beginning of your application. Also, the function CloseHandle does the initialization again to make sure that no pins continue to excite the external circuitry when the application is closed.

When initialization is done the following settings are applied to the device:

- all existing and non-reserved PIC's pins that are available to the user as a general purpose pins are configured as digital inputs, pull-up resistors are disconnected,
- all latch registers are set to zero, which means that if any pin is configured as digital output prior to setting its latch value, it will output 0 V (logic zero),
- internal AD module is turned off,
- internal comparators' module is turned off,
- internal voltage Reference module is turned off,
- both internal PWM modules are turned off,
- internal SPI/I$^2$C module is turned off,
- internal USART module is turned off,
- all settings associated with periodic actions (chapter 19) are cleared.

## 14.6 Performance expectations and variety of coding styles

eProDas applications are diverse in nature as there exists great diversity of natural science experiments and measurements demands. Sometimes you observe a slowly varying signals or static characteristics of some part, where the speed of execution and sample rate is rather non-demanding. In other cases you would want to observe transition response of some high-frequency circuitry or physical system and for that matter you would need prohibitively high sample rates. As it turns out the sample rate of an instrument (among many other parameters, of course) dictates its possible scenarios of usage and the scope within which you can perform experiments in a satisfactory way.

Minimal operational eProDas device costs around 20 € or so in contrast to a high-end professional instrumentation with price tags of several 10 000 € or more. You do not need to be a genius (although you certainly are) to realize that eProDas can offer you only a modest to poor performance in comparison to the world's champions in heavy measurements category.

Further, the precision of sampling frequency and not only its limit is an important parameter of data acquisition equipment. For the majority of eProDas devices the simple quartz crystal oscillator plays role of a time reference and consequently eProDas system cannot compete in time base precision with expensive devices that possess temperature stabilized or oven-controlled quartz clocks.

To complicate things further, let us mention another important but often overlooked sampling parameter: jitter, which denotes a phenomenon where each period of sampling (or some other action) varies slightly from its nominal duration in a random way. This means that the **average** sampling period may be fairly precise but sampling results are still of a poor quality. To achieve low-jitter operation a carefully crafted low noise environment is needed along with high quality low-noise parts. Generally, eProDas system would need to be properly shielded and its power lines filtered, etc. to be able to specify its jitter. We can only say that you cannot expect from a system on a breadboard (like the one in the Figure 5), which is subjected to all hostile influences of its surrounding, to perform as well as expensive professionally shielded, filtered and carefully developed instrument.

Boy, this sounds really depressing but we feel the need to stress these facts out loud to spare you the disappointment at a later time. Not all is that bad, though. The motive for eProDas development is the desire to offer you a system that can exploit your cheap parts to the maximal extent possible with as little programming effort from your side as possible.

### 14.6.1 The philosophy behind the eProDas API

As the world-class developer you know that buying certain parts and doing non-demanding tasks with them is simple and a fun thing to do. However, when you try to explore the capabilities of the very parts to the limits of reality, things begin to complicate and the funny engineering work abruptly changes into a nightmare. eProDas system is no exception in this regard, but we tried to take the burden of complexity to our shoulders (only to a certain extent, of course).

Generally, eProDas API gives you three ways to accomplish the same task (say, doing AD conversion) as a compromise between the performance and the required effort from your side, when coding data acquisition applications. Please read and compare the descriptions that follow to see what suits your needs the best in order not to throw eProDas into thrash can although it might be able to fully fulfil your demands.

#### 14.6.1.1 Execution of isolated task or group of tasks

The first and also the most straightforward way to achieve the desired action is to call the function for the task. For example, to perform AD conversion, call the yet to be described function `ReadInternalADChannel`. Under the hood, this function forms a proper USB packet with your request and sends it to eProDas device, which executes the order and sends the AD result back to you.

The beauty of such approach is its simplicity but the drawback is the disappointingly low efficiency of the process. Namely, it turns out that USB subsystem of your host PC introduces a significant latency (on the order of 1 ms) into the process of delivering an **isolated** USB packet to the device.

Therefore, one packet travels from the host PC to the device and one back from it, which gives us a limit of performing about 500 executions per second (about 3 000 to 4 000 if you take the advice in the chapter 4). The reality is even worse, since eProDas needs some time to process your requests and the figure is further lowered a bit. Despite the discouraging numbers, when measuring static characteristics or monitoring a slowly varying process this may be all that or more than you need, so in such situations there is no need to rush away from this technically inferior approach.

A bit mode decent performance can be achieved by using functions that accomplish a set of similar tasks by still transmitting one USB packet into each direction. An example is reading of a selected set of AD channels with function `ReadSelectedInternalADChannels`. This way only one USB latency is experienced for delivering, say, eight AD results in comparison to eight latencies for achieving the same result by calling the function `ReadInternalADChannel` separately for eight times. Here the increased performance is not accompanied with the increase of application complexity, since only the function's name (and parameters) but not the coding style is changed.

#### 14.6.1.2 Using backbone interpreters

Sometimes you do not need to execute eight AD readings in a row but eight vastly different tasks in a row, like writing to ports, reading of ports, toggling of pin states, reading of AD results, reading of comparators or sending some data to SPI or USART periphery.

Obviously, eProDas cannot provide functions for all possible combinations of these and other tasks that you may come up with (due to the outcomes of the combinatorial theory ☹). Nonetheless, there is a way to execute these tasks more efficiently than isolated function calls would do, if you are willing to pay the price of a bit more complicated application development. Namely, eProDas possesses a set of the so-called backbones that are nothing more than dedicated interpreters that can execute certain primitive actions, like the mentioned readings of ports or AD inputs. You program the activities of these backbones by properly stuffing USB packets with the instructions that these engines know about.

Currently, there are two general-purpose backbones built into eProDas stack: the `Transfer` (section 23.2) and `WaitState` (section 23.3) backbones. In addition, a special `AccessRAM` backbone (section 24.4) exists, but it usable only for hardcore PIC developers (if you are one of us, do not hesitate to exploit this rave feature).

Often you can succeed in executing ten or twenty diverse activities by transmitting a single USB packet, which gives an order of magnitude better performance than the approach of the section 14.6.1.1. The drawback is that you need to build USB packets by yourself, which is a bit tedious in comparison to plain function calls. Still, you do not need to give up on the idea, since eProDas API helps you with the task, and we are assured that after one or two iterations of trial-and-error learning, you are going to find the backbones a rather convenient feature of eProDas system.

#### 14.6.1.3 Using dedicated periodic actions

Now we are coming to the land of fear and horror. To squeeze every last inch of performance out of the poor old PIC, we built special periodic actions (chapter 19) into the eProDas. This feature does not execute commands that travel on-line along USB cable during the experiment but it instead periodically executes the program that is written to the PIC's program memory prior starting the required activities.

During the experiment (or whatever) only the data (excitation values and measurements' results) travel in the USB packets, which maximizes their utilization. One USB packet may hold values for several periods of execution, so that the average latency for delivering data is minimized to the maximal possible extent. In addition, the device utilizes six USB buffers (for each direction of communication) for delivering the packets on time despite the latency of the USB subsystem.

The program execution is triggered periodically by the device itself and without the influence of the USB delays. This way, as low-jitter and precise frequency of execution is achieved as the quality of eProDas hardware permits it. Also, the speed of execution/throughput is significantly increased in comparison to the two previously described approaches. If you intend to sample certain signal to e.g. calculate it's Fourier spectrum, such approach is usually the only proper choice.

To illustrate the benefits, let us reveal that we managed to achieve about 342 kHz operation (342 000 executions of periodic program per second) on a loopback test where one PIC's port outputs some excitation value and another port reads the value to deliver it back to the host PC; this means the actual throughput of 342 kbytes/s into each direction or bidirectional throughput of 684 kbytes/s.

The frequency limitation is not due to the USB bottlenecks but it is imposed by the duration of the periodic program, which needs 2.92 μs or 35 PIC's instructions at 12 MHz for one execution. Only 4 instructions constellate true program (2 for writing and 2 for reading) and the rest is contributed by eProDas automatically for properly handling USB packets. Therefore, by increasing the number of write+read pairs in one period of execution, the efficiency and data throughput increases. By performing four write+read pairs in a single execution cycle (at execution frequency of 150 kHz), we obtained the peak bidirectional USB throughput of about 1.2 Mbytes/s (600 kbytes/s into each direction). This time the limitation is due to the USB throughput, since the program could execute at 255 kHz.

However, bear in mind that these are peak results, which are not always achievable. You need a decent PC for the task (in our case we used AMD Athlon 64, 3200+), so especially laptop computers deliver significantly lower numbers. Further, the USB bus is not predictable to a certain extent, so the peak numbers are not achieved on each run of the test. In practice it is advisory to keep a decent safety margin at least when developing applications for the masses. Also, test them on different computers to see how their horsepower influences the performance limits.

The dark side of periodic actions is that they are significantly more tedious to work with than the previous two approaches. Especially the novice programmers may find this option prohibitively complex, although when the basic principles are mastered the application development is not that hard any more (we think, and some independent testers verify it). If you need the mentioned level of performance that periodic actions deliver, we kindly suggest you to read the material about them several times before giving up on this neat but elaborate idea.

All in all, periodic actions give you significantly more data-acquisition bang for your hard earned bucks than the two previously discussed approaches. For that matter we strongly recommend you to become familiar with them, although there is no need to be forced to use them in each and every case.

# 15 Accessing hardware capabilities of PIC microcontroller

eProDas functions in this chapter enable you to access and exploit PIC's hardware capabilities without being buried by technical details of the chip and USB connection. Further chapters describe purely software implemented higher eProDas level functionality, which enriches these basic features and simplifies eProDas application development even further.

## 15.1 Operations on digital I/O ports

This group of functions focuses on general-purpose digital I/O ports and their operations. The discussion assumes that you are familiar with the functionality of PIC's I/O pins and the associated terminology. If this is not the case, please read appendix A before proceeding with this section.

**Note.** Section 15.1 contains only a subset of all eProDas functions for working with ports. Please, see sections 16.1 and 16.2 for description of many more useful related functions.

### 15.1.1 ConfigurePort

Prototype in C/C++

```
int eProDas_ConfigurePort(unsigned int DeviceIdx, unsigned int Port,
  unsigned char NewTris, unsigned int Action);
```

Prototype in Delphi

```
function eProDas_ConfigurePort(DeviceIdx: LongWord; Port: LongWord; NewTris: Byte;
  Action: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_ConfigurePort(ByVal DeviceIdx As UInteger, ByVal Port As UInteger,
  ByVal NewTris As Byte, ByVal Action As UInteger) As Integer
```

With this function you configure general purpose I/O pins to be either digital inputs or digital outputs by means of setting their associated `TRIS` register to an appropriate value. If certain bit of a `TRIS` register is set (equals 1) the associated pin becomes digital input, otherwise the pin is digital output.

Parameter `Port` specifies which port you intend to configure. The value of 0 means `PORTA`, 1 specifies `PORTB`, etc. Instead of pure numbers you can make your code more readable by using the constants in the Table 2.

| Name | Value |
|---|---:|
| `eProDas_Index_PORTA` | 0 |
| `eProDas_Index_PORTB` | 1 |
| `eProDas_Index_PORTC` | 2 |
| `eProDas_Index_PORTD` | 3 |
| `eProDas_Index_PORTE` | 4 |

**Table 4: Constants for specifying ports.**

Specifying a non-existing port will result in error `eProDas_Error_InvalidPortIndex`.

The lowest eight bits of the parameter `NewTris` specify the entire or partial new `TRIS` value for the port. Exactly how this value is used is determined by the `Action` parameter, which is a bit mask field, where isolated bits specify the requested actions. There exist the following actions, which are specified by the constants in the Table 5.

| Name | Value |
|---|---|
| `eProDas_TRIS_Action_WriteALL` | 256 |
| `eProDas_TRIS_Action_WriteOnes` | 512 |
| `eProDas_TRIS_Action_WriteZeros` | 1024 |
| `eProDas_TRIS_Action_ChangeOnes` | 2048 |

**Table 5: Constants for specifying TRIS actions.**

- Action `TRIS_Action_WriteALL` instructs that the value of the `NewTris` is to be directly written to the `TRIS` register of the specified port.

- In the case of action `TRIS_Action_WriteOnes` only those bits of `NewTris` that equals 1 are copied (logic OR operation). This means that all of the pins that are configured as digital inputs prior to calling this function will preserve their functionality. However, some or all of digital output pins may change their role into digital inputs according to the value `NewTris`.

- Similarly, the action `TRIS_Action_WriteZeros` writes only `NewTris` zeros to the `TRIS` register (logic AND operation), which means that it preserves the digital outputs but digital inputs may switch the role according to the value of the parameter `NewTris`.

- Finally, if you want to change roles of certain pins but preserve the functionality of the others use the action `TRIS_Action_ChangeOnes`. Now, the ones in the `NewTris` parameter specify pins that change the role (logic XOR operation).

Since `Action` parameter is a bit mask field you do not get an error if its value differs from the specifications in the Table 5. If none of the bits from 8 to 11 of the parameter `Action` are set, `TRIS` register will not change. When more than one of these bits is set, the result is unpredictable (i.e. we do not want to specify it in order to preserve the liberty to change the functionality later on).

You may notice that none of the constants in the Table 5 affects any of the lowest eight bits of the `Action` parameter. This opens the possibility to define further actions that may require additional bit mask residing in the lowest eight bits of the value. Suggestions are welcome.

**Important!** This function does not allow you to configure pins in a way that would conflict with the functionality of other on-chip peripherals. For example, if certain pin is configured as one of the inputs of the AD converter, you will not be able to configure it as digital output. Enabled peripherals always take precedence over pure digital I/O functionality, since if you put your effort into enabling some turned-off-by-default module you definitively intend to use it. Therefore, when it seems that the function `ConfigurePort` refuses to comply with your orders, do not get mad. Instead check the conflicts with currently enabled peripherals.

**Remarks.** Internally, this function relies on function `ConfigurePorts` (see below), which always reconfigures all `TRIS` registers. Ports, which are not referred to by the `Port` parameter, are technically reconfigured with the same `TRIS` value. To achieve this functionality, the old values of all `TRIS` registers are read prior taking any action. Then user requested changes are applied to the read values. After that, conflicts with enabled peripherals are resolved and the resulted values are written back to the physical `TRIS` registers.

This seems a rather inefficient way to do the job. However, this elaborate procedure can rely on only one central place, the function `ConfigurePorts`, to take care of checking the settings. In complicated cases it is beneficiary that function `ConfigurePorts` knows all `TRIS` values and other aspects of device configuration to be able to properly resolve the potential conflicts.

If you want a more efficient way to configure ports but without any protection, there exist two functions `ConfigurePortUnsafe` (section 15.1.5) and `ConfigurePortsUnsafe` (section 15.1.6) that will hopefully fulfil your dreams.

### 15.1.2 ConfigurePorts

Prototype in C/C++

```
int eProDas_ConfigurePorts(unsigned int DeviceIdx, unsigned char *NewTrises,
  unsigned int *Actions);
```

Prototype in Delphi

```
function eProDas_ConfigurePorts(DeviceIdx: LongWord; NewTrises: PByte;
  Actions: PLongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_ConfigurePorts(ByVal DeviceIdx As UInteger,
  ByRef NewTrises As Byte, ByRef Actions As UInteger) As Integer
```

Sometimes you want to reconfigure all ports at once instead of just one of them. On such occasions the function `ConfigurePorts` may be handy. Parameter `NewTrises` is a pointer to an array of new `TRIS` values and similarly parameter `Actions` points to an array of action values. Both of them work in precisely the same way as they do with the function `ConfigurePort`. The first entries in both arrays take care of reconfiguring `PORTA`, the second ones are for the `PORTB`, etc. Make sure that the arrays have one entry for each port or an access violation error will result.

**Note 1.** As previously stated the function `ConfigurePort` relies on this function to do the hard work. Hence, it is strongly recommended to read section 15.1.1 with some important explanations.

**Note 2.** Although you rarely need to configure exactly all ports, this is not the reason to avoid using this function. Just take care that you specify no action for the ports that must preserve their configuration and you can safely use this function instead of slower (see section 14.6.1.1) multiple calls to the function `ConfigurePort`.

### 15.1.3 ConfigurePinAsOutput

Prototype in C/C++

```
int eProDas_ConfigurePinAsOutput(unsigned int DeviceIdx, unsigned int Port,
  unsigned int Pin, unsigned int InitialState);
```

Prototype in Delphi

```
function eProDas_ConfigurePinAsOutput(DeviceIdx: LongWord; Port: LongWord;
  Pin: LongWord; InitialState: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_ConfigurePinAsOutput(ByVal DeviceIdx As UInteger,
  ByVal Port As UInteger, ByVal Pin As UInteger, ByVal InitialState As UInteger)
  As Integer
```

Frequently you want to configure only one pin as digital output and determine its initial state. Such example is configuration of pin for connecting *ChipSelect* signal of some external circuitry to a PIC. In such cases you also want to make sure that the circuitry is not accidentally activated by improper state of the pin. This function is a handy shortcut to achieve the described behaviour.

The parameters `Port` (Table 4) and `Pin` (from 0 to 7) specify the PIC's pin in question whereas the parameter `InitialState` determines the initial state of the pin (any non-zero value results in pin state logic 1). The function changes the port's latch register (pin state) before it configures pin as digital output to prevent any unnecessary state transitions on the pin.

**Note 1.** If `Port` is greater than or equal to 1 000 this function does nothing at all besides returning a `SUCCESS` code. This is an intended behaviour to simplify application coding in cases such as conditional *ChipSelect* handling.

**Note 2.** The configuration and state of all other pins of the same port is preserved by this function and no spikes on other pins are produced during the execution.

### 15.1.4 ConfigurePinAsInput

Prototype in C/C++

```
int eProDas_ConfigurePinAsInput(unsigned int DeviceIdx, unsigned int Port,
  unsigned int Pin);
```

Prototype in Delphi

```
function eProDas_ConfigurePinAsInput(DeviceIdx: LongWord; Port: LongWord;
  Pin: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_ConfigurePinAsInput(ByVal DeviceIdx As UInteger,
  ByVal Port As UInteger, ByVal Pin As UInteger) As Integer
```

Contrary to the previously described function we now want to convert only one specified pin into digital input. The parameters work in the same way as they do with the previous function.

### 15.1.5 ConfigurePortUnsafe

Prototype in C/C++

```
int eProDas_ConfigurePortUnsafe(unsigned int DeviceIdx, unsigned int Port,
  unsigned char NewTris);
```

Prototype in Delphi

```
function eProDas_ConfigurePortUnsafe(DeviceIdx: LongWord; Port: LongWord;
  NewTris: Byte):integer;
```

Prototype in VisualBasic

```
Function eProDas_ConfigurePortUnsave(ByVal DeviceIdx As UInteger,
  ByVal Port As UInteger, ByVal NewTris As Byte) As Integer
```

The *unsafe* set of functions accompanies the previously described *safe* functions for configuring I/O ports, however they are less safe (hence the name ☺) but more efficient to use, i.e. they execute faster than their safe counterparts.

The function `ConfigurePortUnsafe` serves the same purpose as the function `ConfigurePort` does. The difference is that `ConfigurePortUnsafe` merely copies value `NewTris` to an appropriate `TRIS` register. Greater efficiency stems from the fact that you cannot specify other actions on `TRIS` register besides plain copy, which means that the old `TRIS` value does not need to be read before a write operation can take place. Further, the `TRIS` value is not checked for possible clashes with other enabled modules. You have been warned…

**Note.** Despite the previous warning, the unsafe functions can be very useful, for example when you implement/emulate microprocessor data bus (possibly by combining several PIC's ports into a wide bus), which needs to be reconfigured frequently to reflect the intended direction of dataflow.

### 15.1.6 ConfigurePortsUnsafe

Prototype in C/C++

```
int eProDas_ConfigurePortsUnsafe(unsigned int DeviceIdx, unsigned char *NewTrises);
```

Prototype in Delphi

```
function eProDas_ConfigurePortsUnsafe(DeviceIdx: LongWord;
  NewTrises: PByte):integer;
```

Prototype in VisualBasic

```
Function eProDas_ConfigurePortsUnsave(ByVal DeviceIdx As UInteger,
  ByRef NewTrises As Byte) As Integer
```

With this function you copy new values into all `TRIS` registers at once. Parameter `NewTrises` points to an array of 8-bit values, which contains new `TRIS` values. This array must be large enough to hold all `TRIS` values or Access Violation error will result. Again, new values are faithfully copied over the old ones without any consistency checking.

### 15.1.7 XOR_PortsUnsafe

Prototype in C/C++

```
eProDasDLL_API int eProDas_XOR_PortsUnsafe(unsigned int DeviceIdx,
  unsigned char *XOR_Trises);
```

Prototype in Delphi

```
function eProDas_XOR_PortsUnsafe(DeviceIdx: LongWord; XOR_Trises: Byte):integer;
```

Prototype in VisualBasic

```
Function eProDas_XOR_PortsUnsafe(ByVal DeviceIdx As UInteger,
  ByVal XOR_Trises As Byte) As Integer
```

With this function you perform a bitwise XOR operation between the current TRIS-es and the values that you provide through the array `XOR_Trises`. XOR operation is handy, when you want to switch the role of certain pins without affecting the rest of ports' configuration.

### 15.1.8 ConfigureSelectedPortsUnsafe

Prototype in C/C++

```
int eProDas_ConfigureSelectedPortsUnsafe(unsigned int DeviceIdx,
  unsigned int PortFlags, unsigned char *NewTrises);
```

Prototype in Delphi

```
function eProDas_ConfigureSelectedPortsUnsafe(DeviceIdx: LongWord;
  PortFlags: LongWord; NewTrises: PByte):integer;
```

Prototype in VisualBasic

```
Function eProDas_ConfigureSelectedPortsUnsafe(ByVal DeviceIdx As UInteger,
  ByVal PortFlags As UInteger, ByRef NewTrises As Byte) As Integer
```

Instead of reconfiguring all ports, which often does not make sense, you can reconfigure only a selected subset of them. The selection is specified by setting the appropriate bits of the bit field `PortFlags`; bit 0 for port A…, bit 4 for port E. The values to be written to the `TRIS` registers are read from the array `NewTrises` in the ascending order of port name (index).

Only values for the selected ports are read, so by specifying `PortFlags` as 19 (1+2+16 for ports A, B and E), the first three values in the array `NewTrises` would be copied to `TRIS` registers of port A, B and E, in that order.

### 15.1.9 XOR_SelectedPortsUnsafe

Prototype in C/C++

```
int eProDas_XOR_SelectedPortsUnsafe(unsigned int DeviceIdx, unsigned int PortFlags,
  unsigned char *XOR_Trises);
```

Prototype in Delphi

```
function eProDas_XOR_SelectedPortsUnsafe(DeviceIdx: LongWord;
  PortFlags: LongWord; XOR_Trises: PByte):integer;
```

Prototype in VisualBasic

```
Function eProDas_XOR_SelectedPortsUnsafe(ByVal DeviceIdx As UInteger,
  ByVal PortFlags As UInteger, ByRef XOR_Trises As Byte) As Integer
```

An analogous function with the previous one, except that the user specified values are not directly copied into selected `TRIS` registers but they are instead bitwise XOR-ed together.

### 15.1.10 Configure_XOR_SelectedPortsUnsafe

Prototype in C/C++

```
int eProDas_Configure_XOR_SelectedPortsUnsafe(unsigned int DeviceIdx,
  unsigned int WriteFlags, unsigned int XORFlags, unsigned char *New_XOR_Trises);
```

Prototype in Delphi

```
function eProDas_Configure_XOR_SelectedPortsUnsafe(DeviceIdx: LongWord;
  WriteFlags: LongWord; XORFlags: LongWord; New_XOR_Trises: PByte):integer;
```

Prototype in VisualBasic

```
Function eProDas_Configure_XOR_SelectedPortsUnsafe(ByVal DeviceIdx As UInteger,
  ByVal WriteFlags As UInteger, ByVal XORFlags As UInteger,
  ByRef New_XOR_Trises As Byte) As Integer
```

With this function you can do both tasks of writing and XOR-ing the `TRIS` registers simultaneously. Which ports are affected, is specified separately for both actions through the parameters `WriteFlags` and `XORFlags`. All write operations are done first and XOR-ing after that, which is reflected by the sequence of readings of the values from the array `New_XOR_Trises`.

**Note.** If you both write to and XOR certain port, the intermediate written value is applied to the physical TRIS register for a certain amount of time before XOR operation also takes place. This means that according to your values, certain pins may be configured in an undesired way during the time of function execution with various possible consequences. To eliminate the danger of side effects like burned chips, we strongly advise you not to exercise both operations on the same port, unless you know what you are doing (and there is a possibility that you do not).

### 15.1.11 ConfigurePinAsOutputUnsafe and ConfigurePinAsInputUnsafe

Prototype in C/C++

```
int eProDas_ConfigurePinAsOutputUnsafe(unsigned int DeviceIdx, unsigned int Port,
  unsigned int Pin, unsigned int InitialState);

int eProDas_ConfigurePinAsInputUnsafe(unsigned int DeviceIdx, unsigned int Port,
  unsigned int Pin);
```

Prototype in Delphi

```
function eProDas_ConfigurePinAsOutputUnsafe(DeviceIdx: LongWord; Port: LongWord;
  Pin: LongWord; InitialState: LongWord):integer;

function eProDas_ConfigurePinAsInputUnsafe(DeviceIdx: LongWord; Port: LongWord;
  Pin: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_ConfigurePinAsOutputUnsafe(ByVal DeviceIdx As UInteger, ByVal Port
  As UInteger, ByVal Pin As UInteger, ByVal InitialState As UInteger)
  As Integer

Function eProDas_ConfigurePinAsInputUnsafe(ByVal DeviceIdx As UInteger,
  ByVal Port As UInteger, ByVal Pin As UInteger) As Integer
```

If you managed to follow this lengthy sequel about ports' configuration, then these two functions should be self-descriptive. They work in the same way as the two functions without literal *Unsafe* in their names, but now there is no consistency checking and there is gain in the speed of execution.

### 15.1.12 ReadPort

Prototype in C/C++

```
int eProDas_ReadPort(unsigned int DeviceIdx, unsigned int Port,
  unsigned char &Value);
```

Prototype in Delphi

```
function eProDas_ReadPort(DeviceIdx: LongWord; Port: LongWord;
  var Value: Byte):integer;
```

Prototype in VisualBasic

```
Function eProDas_ReadPort(ByVal DeviceIdx As UInteger, ByVal Port As UInteger,
  ByRef Value As Byte) As Integer
```

This function reads digital state of pins of the port, specified by the parameter `Port`, by means of reading the associated `PORT` register of the PIC. The retrieved value is stored in the parameter `Value`.

The state of pins is realistic, which means that if a pin is configured as digital output but its state is overridden by the strong outside voltage source (which will probably destroy your chip in a short time), you will get the actual happening.

### 15.1.13 ReadPorts

Prototype in C/C++

```
int eProDas_ReadPorts(unsigned int DeviceIdx, unsigned char *Values);
```

Prototype in Delphi

```
function eProDas_ReadPorts(DeviceIdx: LongWord; Values: PByte):integer;
```

Prototype in VisualBasic

```
Function eProDas_ReadPorts(ByVal DeviceIdx As UInteger, ByRef Values As Byte)
  As Integer
```

With this function you can read state of all ports at once into the array that is pointed to by the parameter `Values`. Although you rarely need to read all ports, you may find this function handy, since it is more efficient to execute than double call to function `ReadPort`. Namely, with the latter you initiate a separate USB transfer for each reading of port, whereas the function `ReadPorts` transfers all values at once with minimal overhead in comparison to reading a single port.

### 15.1.14 ReadSelectedPorts

Prototype in C/C++

```
int eProDas_ReadSelectedPorts(unsigned int DeviceIdx, unsigned int PortFlags,
  unsigned char *Values);
```

Prototype in Delphi

```
function eProDas_ReadSelectedPorts(DeviceIdx: LongWord; PortFlags: LongWord;
  Values: PByte):integer;
```

Prototype in VisualBasic

```
Function eProDas_ReadSelectedPorts(ByVal DeviceIdx As UInteger,
  ByVal PortFlags As UInteger, ByRef Values As Byte) As Integer
```

Still better approach in the majority of cases is to specify a subset of ports that you are interested in reading. The specification of ports is done as usual by setting the appropriate bits of `PortFlags` bitmask. The sequence of reading is from port A to port E and the resulted values are written according to this sequence into the user allocated array `Values`.

**Note.** Ports A, C and E do not have (available) the last two pins (6 and 7), which this function exploits for delivering the current state of comparators (section 15.3) in these two bits (comparator 1 in bit 6 and comparator 2 in bit 7). Therefore, if you read any of the three ports and you are also interested in the state of comparators, you can spare execution of one command.

### 15.1.15 WritePort

Prototype in C/C++

```
int eProDas_WritePort(unsigned int DeviceIdx, unsigned int Port,
  unsigned char NewValue);
```

Prototype in Delphi

```
function eProDas_WritePort(DeviceIdx: LongWord; Port: LongWord;
  NewValue: Byte):integer;
```

Prototype in VisualBasic

```
Function eProDas_WritePort(ByVal DeviceIdx As UInteger, ByVal Port As UInteger,
  ByVal NewValue As Byte) As Integer
```

With this function you write `NewValue` into the data latch (register `LAT`) of a port, that you specify with the parameter `Port`. The written value immediately affects the state of pins that are configured as digital outputs.

**Note.** The value `NewValue` is written into the entire data latch and it affects all latch bits; even those that belong to pins that are configured as digital inputs. Although physical pins are not influenced by the new value immediately, their state will be influenced if the role of a pin changes to digital output somewhere in the future.

### 15.1.16 WritePorts

Prototype in C/C++

```
int eProDas_WritePorts(unsigned int DeviceIdx, unsigned char *NewValues);
```

Prototype in Delphi

```
function eProDas_WritePorts(DeviceIdx: LongWord; NewValues: PByte):integer;
```

Prototype in VisualBasic

```
Function eProDas_WritePorts(ByVal DeviceIdx As UInteger, ByRef NewValues As Byte)
  As Integer
```

Use this function to write values of the array `NewValues` to the latches of all ports. Although more efficient than executing two separate `WritePort` functions we cannot recommend you to use this function on a regular basis as we favoured usage of `ReadPorts` over two or more separate `ReadPort` calls. Namely, reading of ports is a safe operation without side effects, however writing to them and changing state of pins certainly is not.

### 15.1.17 WriteSelectedPorts

Prototype in C/C++

```
int eProDas_WriteSelectedPorts(unsigned int DeviceIdx, unsigned int PortFlags,
  unsigned char *NewValues);
```

Prototype in Delphi

```
function eProDas_WriteSelectedPorts(DeviceIdx: LongWord; PortFlags: LongWord;
  NewValues: PByte):integer;
```

Prototype in VisualBasic

```
Function eProDas_WriteSelectedPorts(ByVal DeviceIdx As UInteger,
  ByVal PortFlags As UInteger, ByRef NewValues As Byte) As Integer
```

With this function you can specify the arbitrary subset of ports to be written to. Again, the specification of ports is done by setting the appropriate bits of `PortFlags` bitmask. The sequence of writing is from port A to port E and the values in the array `NewValues` must respect this order.

### 15.1.18 XOR_SelectedPorts

Prototype in C/C++

```
int eProDas_XOR_SelectedPorts(unsigned int DeviceIdx, unsigned int PortFlags,
  unsigned char *XOR_Values);
```

Prototype in Delphi

```
function eProDas_XOR_SelectedPorts(DeviceIdx: LongWord; PortFlags: LongWord;
  XOR_Values: PByte):integer;
```

Prototype in VisualBasic

```
Function eProDas_XOR_SelectedPorts(ByVal DeviceIdx As UInteger, ByVal PortFlags As
  UInteger, ByRef XOR_Values As Byte) As Integer
```

The same functionality as with the previous function, except that the current ports' latch values are bitwise XOR-ed with a user supplied values through the array `XOR_Values` instead of being directly written to the latch.

**Note.** The function for XOR-ing an isolated port is described in section 16.1.1.

### 15.1.19 ReadWriteSelectedPorts

Prototype in C/C++

```
int eProDas_ReadWriteSelectedPorts(unsigned int DeviceIdx, unsigned int ReadFlags,
  unsigned int WriteFlags, unsigned char *ReadValues, unsigned char *WriteValues);
```

Prototype in Delphi

```
function eProDas_ReadWriteSelectedPorts(DeviceIdx: LongWord; ReadFlags: LongWord;
  WriteFlags: LongWord; ReadValues: PByte; WriteValues: PByte):integer;
```

Prototype in VisualBasic

```
Function eProDas_ReadWriteSelectedPorts(ByVal DeviceIdx As UInteger,
  ByVal ReadFlags As UInteger, ByVal WriteFlags As UInteger,
  ByRef ReadValues As Byte, ByRef WriteValues As Byte) As Integer
```

Function `ReadWriteSelectedPorts` combines the functionality of the functions `ReadSelectedPorts` and `WriteSelectedPorts`. The arbitrary and independent sets of ports for reading and writing can be specified by setting the appropriate bits of the two bit fields `ReadFlags` and `WriteFlags`. The read values are stored in the user allocated array `ReadValues`, whereas the values to be written to the ports are obtained from the user provided array `WriteValues`.

**Note 1.** The same port can be specified for both operations and this case often makes sense.

**Note 2.** Ports A, C and E do not have (available) the last two pins (6 and 7), which this function exploits for delivering the current state of comparators (section 15.3) in these two bits (comparator 1 in bit 6 and comparator 2 in bit 7). Therefore, if you read any of the three ports and you are also interested in the state of comparators, you can spare execution of one command.

### 15.1.20 Read_XOR_SelectedPorts

Prototype in C/C++

```
int eProDas_Read_XOR_SelectedPorts(unsigned int DeviceIdx, unsigned int ReadFlags,
  unsigned int XOR_Flags, unsigned char *ReadValues, unsigned char *XOR_Values);
```

Prototype in Delphi

```
function eProDas_Read_XOR_SelectedPorts(DeviceIdx: LongWord; ReadFlags: LongWord;
  XOR_Flags: LongWord; ReadValues: PByte; XOR_Values: PByte):integer;
```

Prototype in VisualBasic

```
Function eProDas_Read_XOR_SelectedPorts(ByVal DeviceIdx As UInteger,
  ByVal ReadFlags As UInteger, ByVal XOR_Flags As UInteger, ByRef ReadValues As
  Byte, ByRef XOR_Values As Byte) As Integer
```

Similar functionality as in the case of the previous function except that writing has been replaced with XOR-ing of latch values.

**Note.** Ports A, C and E do not have (available) the last two pins (6 and 7), which this function exploits for delivering the current state of comparators (section 15.3) in these two bits (comparator 1 in bit 6 and comparator 2 in bit 7). Therefore, if you read any of the three ports and you are also interested in the state of comparators, you can spare execution of one command.

### 15.1.21 SetPin, ClearPin, TogglePin, TogglePinTwice

Prototypes in C/C++

```
int eProDas_SetPin(unsigned int DeviceIdx, unsigned int Port, unsigned int Pin);

int eProDas_ClearPin(unsigned int DeviceIdx, unsigned int Port, unsigned int Pin);

int eProDas_TogglePin(unsigned int DeviceIdx, unsigned int Port, unsigned int Pin);

int eProDas_TogglePinTwice(unsigned int DeviceIdx, unsigned int Port,
  unsigned int Pin);
```

Prototypes in Delphi

```
function eProDas_SetPin(DeviceIdx: LongWord; Port: LongWord;
  Pin: LongWord):integer;

function eProDas_ClearPin(DeviceIdx: LongWord; Port: LongWord;
  Pin: LongWord):integer;

function eProDas_TogglePin(DeviceIdx: LongWord; Port: LongWord;
  Pin: LongWord):integer;

function eProDas_TogglePinTwice(DeviceIdx: LongWord; Port: LongWord;
  Pin: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_SetPin(ByVal DeviceIdx As UInteger, ByVal Port As UInteger,
  ByVal Pin As UInteger) As Integer

Function eProDas_ClearPin(ByVal DeviceIdx As UInteger, ByVal Port As UInteger,
  ByVal Pin As UInteger) As Integer

Function eProDas_TogglePin(ByVal DeviceIdx As UInteger, ByVal Port As UInteger,
  ByVal Pin As UInteger) As Integer

Function eProDas_TogglePinTwice(ByVal DeviceIdx As UInteger,
  ByVal Port As UInteger, ByVal Pin As UInteger) As Integer
```

Here comes another set of (hopefully) useful functions, which this time works on isolated pins instead of the whole port(s). Function `SetPin` puts the specified pin in the state logic 1, whereas `ClearPin` puts it in the state logic 0 and `TogglePin` toggles the current state. The function `TogglePinTwice` toggles pin state twice in a short time (about 83 ns apart) to produce a pulse that is convenient e.g. for triggering clock signal or to make external latch transparent for a short period of time.

## 15.2 Operations on internal AD converter module

Our PIC possesses a built-in 10-bit resolution analog-to-digital (AD) converter with 13 multiplexed analog inputs. This AD converter is referred to as “internal” one to distinguish it from other AD converters that various eProDas devices may have instead or in addition to the internal one. The literal “InternalAD” is a part of all function names that operate on the PIC’s internal AD converter.

**Note.** Additional functions that operate on internal AD converter are described in section 16.3.

### 15.2.1 ConfigureInternalAD

Prototype in C/C++

```
int eProDas_ConfigureInternalAD(unsigned int DeviceIdx,
  unsigned int NumberOfADPorts, unsigned int AcquisitionTimeCode,
  unsigned int VrefPlus, unsigned int VrefMinus);
```

Prototype in Delphi

```
function eProDas_ConfigureInternalAD(DeviceIdx: LongWord;
  NumberOfADPorts: LongWord; AcquisitionTimeCode: LongWord;
  VrefPlus: LongWord; VrefMinus: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_ConfigureInternalAD(ByVal DeviceIdx As UInteger,
  ByVal NumberOfADPorts As UInteger, ByVal AcquisitionTimeCode As UInteger,
  ByVal VrefPlus As UInteger, ByVal Vrefminus As UInteger) As Integer
```

As the name reveals, with this function you configure internal AD module before you can use it. Parameter `NumberOfADPorts` specifies number of multiplexed AD channels that you intend to use. The value must not be larger than 13 or an error `eProDas_Error_InvalidNumberOfADChannels` will result. Specifying zero number of AD ports shuts down the AD module.

AD channels are mapped to the physical pins according to the Table 6; see also Figure 2 on page 21.

| AD channel | Physical pin |
|---|---|
| AN0 | RA0 |
| AN1 | RA1 |
| AN2 | RA2 |
| AN3 | RA3 |
| AN4 | RA5 |
| AN5 | RE0 |
| AN6 | RE1 |
| AN7 | RE2 |
| AN8 | RB2 |
| AN9 | RB3 |
| AN10 | RB1 |
| AN11 | RB4 |
| AN12 | RB0 |

**Table 6: Mapping between internal AD channels and physical pins.**

For example, if you configure AD module to have 7 AD channels then pins RA0…RA3, RA5, RE0 and RE1 will become analog inputs. You can only select the number of AD channels but not their arbitrary pins. Therefore, you cannot select e.g. pins RA0 and RA5 to be analog inputs but preserve RA1, RA2 and RA3 as digital pins. If any pins, which are about to become analog input, are configured as digital outputs prior to calling this function, then they are first reconfigured as digital inputs in order not to excite the pin which is (probably) driven by external analog circuitry.

The parameter `AcquisitionTimeCode` is designed for torturing people with technical details. The acquisition time is the time between the selection of analog input and the moment when AD conversion starts. A somewhat comprehensive explanation of the reason why there should be a delay between the two events can be found in the appendix B. If you are not interested in the material just make sure that the acquisition time is long at least 8 µs and you will be fine in most (but not all) cases.

Now, the PIC can help you with this. Namely, whenever you instruct it to start AD conversion it can automatically wait a certain amount of time before AD conversion actually starts. This delay, the acquisition time, can be selected as a certain multiple of a period of AD clock which equals (48 MHz)/64 = 1.333 µs. Select the length of the acquisition time according to the Table 7.

| **AcquisitionTimeCode** | **AD clock periods** | **Acquisition time** [µs] |
|---|---:|---:|
| 0 | 0 | 0 |
| 1 | 2 | 2,7 |
| 2 | 4 | 5,3 |
| 3 | 6 | 8,0 |
| 4 | 8 | 11 |
| 5 | 12 | 16 |
| 6 | 16 | 21 |
| 7 | 20 | 27 |

**Table 7: Mapping between `AcquisitionTimeCode` and resulting acquisition time.**

Finally, parameters `VrefPlus` and `VrefMinus` select how the reference voltage of AD converter is obtained. In the cases where you do not need great AD accuracy you can simply use both power supply rails for the task. The advantage of such choice is cost and space saving. However, if the power supply voltage value is not precisely what you expect, the accuracy of AD conversion will be decreased. The exception may be the ratio-metric measurements.

You may control plus and minus reference voltage sources independently. When `VrefPlus` is zero the positive voltage reference is connected to VDD, whereas in the opposite case you bring the positive pin of external (precision) source like band-gap reference of your choice to the pin RA3. Similarly, when parameter `VrefMinus` equals zero the negative voltage reference is connected to VSS pin, whereas the pin RA2 fulfils this role in the opposite case.

### 15.2.2 ConfigureInternalAD_Advanced

Prototype in C/C++

```
int eProDas_ConfigureInternalAD_Advanced(unsigned int DeviceIdx,
  unsigned int NumberOfADPorts, unsigned int AcquisitionTimeCode,
  unsigned int VrefPlus, unsigned int VrefMinus, unsigned int Justification,
  unsigned int FullAccuracy);
```

Prototype in Delphi

```
function eProDas_ConfigureInternalAD_Advanced (DeviceIdx: LongWord;
  NumberOfADPorts: LongWord; AcquisitionTimeCode: LongWord;
  VrefPlus: LongWord; VrefMinus: LongWord; Justification: LongWord;
  FullAccuracy: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_ConfigureInternalAD_Advanced (ByVal DeviceIdx As UInteger,
  ByVal NumberOfADPorts As UInteger, ByVal AcquisitionTimeCode As UInteger,
  ByVal VrefPlus As UInteger, ByVal Vrefminus As UInteger,
  ByVal Justification As UInteger, ByVal FullAccuracy As UInteger) As Integer
```

This function works the same as the *non-advanced* version does, except that is possesses two additional parameters by which you can configure some rarely used aspects of internal AD module.

**Note.** The additional possibilities only make sense if you intend to use the AD module during periodic actions (chapter 19). If this is not the case, ignore this function and use the previous one instead.

With parameter `Justification` you select the alignment of the AD result. A non-zero value of this parameter selects right justification whereas zero value results in left justification. Now, what are we babbling about?

Since AD converter delivers 10-bit result but your computer prefers to chew numbers in 8-bit pieces at a time, the 10-bit result is in fact stored in a 16-bit number, where unused 6 bits are always set to zero. With right justification, which is a natural *non-advanced* way of doing things, the result occupies 10 the rightmost bits in the 16-bit result. And off course with left justification the result is contained in the 10 leftmost bits.

Generally, you need right justification so that the delivered 16-bit number equals the 10-bit result. However, if you need only 8-bit precision you can more easily process the left justified result since 8 most significant bits are contained in a single byte (the second one) and you can put the result in an 8-bit storage space without bit shift operations.

This makes sense during high-frequency and high-throughput periodic actions, which are taught how to read the most significant 8-bits of the left aligned AD result, by means of which you save one precious byte of USB packet.

The `FullAccuracy` is a rather tricky parameter with a bad reputation. You will make yourself a big favour if you simply skip reading the following explanation and make sure that this parameter has a non-zero value.

Each AD in this world requires a certain amount of time to correctly perform AD conversion and the PIC's internal AD is no exception. The inner working of AD module requires a period of 11 AD clocks to deliver result. The frequency of AD clock is under the control of the user, which may select 7 different values. Except for one case, which is not relevant to eProDas, the selected frequency is obtained by dividing the microprocessor's clock frequency (48 MHz) with a divider, which can be one of the following: 2, 4, 8, 16, 32 or 64.

For example, if you select 64 as a divider, the frequency of AD clock will be (48 MHz)/64 = 750 kHz. Since AD conversion takes 11 AD clocks, the conversion time is: 11/(750 kHz) = 14.667 µs. By selecting a smaller divider, say 32, you get faster AD conversion with a period of 7.333 µs.

You cannot do this arbitrarily, however, since the AD module loses the accuracy if the AD clock period is shorter than 0.7 µs, which means that at 48 MHz of processor clock, a divider of 64 is the only permissible choice. But there is a catch. According to the datasheet, if the microprocessor clock equals 40 MHz, which is slightly less than 48 MHz, a divider of 32 would be permissive. Now, if you do not need 10-bit accuracy and you are willing to experiment a bit, you may try to use the divider of 32 even at 48 MHz processor clock, which you do by selecting a zero value of the parameter `FullAccuracy`.

There is no guarantee about the resulting accuracy under this condition. It could be the whole 10 bits (probably not, otherwise Microchip would say so and make the chip more attractive) down to 0 bit. This idea of over clocking is NOT proposed in a datasheet but it is instead a result of a sick mind of the author of this document. If you intend to do anything serious with eProDas device, do not try to be smart and stick with the original documentation.

Further, even if your particular instance of integrated circuit seems to work fine, the one that is in a possession by your neighbour or even roommate may not work at all. If you do not obey with the datasheet roles you are completely on your own. Good luck and do not forget the warning.

### 15.2.3 AcquireInternalADChannel

Prototype in C/C++

```
int eProDas_AcquireInternalADChannel(unsigned int DeviceIdx, unsigned int ADPort);
```

Prototype in Delphi

```
function eProDas_AcquireInternalADChannel(DeviceIdx: LongWord;
  ADPort: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_AcquireInternalADChannel(ByVal DeviceIdx As UInteger,
  ByVal ADPort As UInteger) As Integer
```

With this function you acquire or select AD channel to be the source for the next AD conversion. As soon as the channel is acquired the acquisition period starts. If you know in advance which channel will be the target for the next AD conversion but you do not know when the conversion will take place (i.e. you are waiting for a trigger, etc.) simply pre-acquire the channel with this function, by means of which the AD conversion can start immediately upon the occurrence of the event that you are waiting for and there is no need to cope with that pesky parameter `AcquisitionTimeCode`.

Parameter `ADPort` selects the channel according to the Table 6. If its value is greater than 12 the error `eProDas_Error_InvalidADChannel` will result. This function does not protect you from selecting a channel that is not configured as AD input. For example, if you configure AD module to have 5 analog inputs, but you try to pre-acquire channel 8, no error will be reported. The datasheet does not mention what happens in this circumstance, but we assume that the worst thing possible is delivered gibberish instead of a meaningful AD result.

### 15.2.4 ReadInternalADChannel

Prototype in C/C++

```
int eProDas_ReadInternalADChannel(unsigned int DeviceIdx, unsigned int &Result);
```

Prototype in Delphi

```
function eProDas_ReadInternalADChannel(DeviceIdx: LongWord;
  var Result: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_ReadInternalADChannel(ByVal DeviceIdx As UInteger,
  ByRef Result As UInteger) As Integer
```

This function starts the actual AD conversion on a previously acquired AD channel. Before the AD conversion process starts the PIC microcontroller automatically inserts a delay that you selected by the parameter `AcquisitionTimeCode` when you configured AD module.

When you initiate the AD conversion with this function the settings of acquisition time are irrelevant, since you cannot send two consecutive commands through the USB connection fast enough that there would be less than 8 µs between the execution of commands `AcquireInternalADChannel` and `ReadInternalADChannel`. However, proper specification of `AcquisitionTimeCode` is important for the next functions as well as for periodic actions (chapter 19).

The result of AD conversion is returned in the `Result` field, which is 32-bits wide, so the upper 22 bits are always 0 (the upper 16 bits if you have selected left justification of the result with the function `ConfigureInternalAD_Advanced`, section 15.2.2).

### 15.2.5 AcquireAndReadInternalADChannel

Prototype in C/C++

```
int eProDas_AcquireAndReadInternalADChannel(unsigned int DeviceIdx,
  unsigned int ADPort, unsigned int &Result);
```

Prototype in Delphi

```
function eProDas_AcquireAndReadInternalADChannel(DeviceIdx: LongWord;
  ADPort: LongWord; var Result: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_AcquireAndReadInternalADChannel(ByVal DeviceIdx As UInteger,
  ByVal ADport As UInteger, ByRef Result As UInteger) As Integer
```

This function covers both just described functionalities. It first acquires the selected AD channel (parameter `ADPort`) and then does the AD conversion on it. Here the configuration of acquisition time is important.

### 15.2.6 ReadAndAcquireInternalADChannel

Prototype in C/C++

```
int eProDas_ReadAndAcquireInternalADChannel(unsigned int DeviceIdx,
  unsigned int &Result, unsigned int ADPortToAcquireAfter);
```

Prototype in Delphi

```
function eProDas_ReadAndAcquireInternalADChannel(DeviceIdx: LongWord;
  var Result: LongWord; ADPortToAcquireAfter: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_ReadAndAcquireInternalADChannel(ByVal DeviceIdx As UInteger,
  ByRef Result As UInteger, ByVal ADPortToAcquireAfter As UInteger) As Integer
```

In some circumstances you may find useful the possibility to perform the previously described functionality in the opposite order, which is to first do the AD conversion and to select the next AD channel after that. This way you start the acquisition time on the next-to-be read channel immediately after the completion of AD conversion on the current channel, so you may carelessly wait for the proper moment to trigger the next conversion.

### 15.2.7 ReadSelectedInternalADChannels

Prototype in C/C++

```
int ePröDas_ReadSelectedInternalADChannels(unsigned int DeviceIdx,
  unsigned int ChannelFlags, unsigned int *Results);
```

Prototype in Delphi

```
function eProDas_ReadSelectedInternalADChannels(DeviceIdx: LongWord;
  ChannelFlags: LongWord; Results: PLongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_ReadSelectedInternalADChannels(ByVal DeviceIdx As UInteger,
  ByVal ChannelFlags As UInteger, ByRef Results As UInteger) As Integer
```

Use this function to read arbitrary subset of internal AD channels. Channels are specified by setting bits of a bitfield `ChannelFlags`. Results are delivered in the ascending order of the channel number.

## 15.3 Operations on internal comparators' module

Within the PIC there are two voltage comparators that you may utilize in your applications for which eProDas API provides a couple of functions. Again, their names contain the literal "Internal" to distinguish internal PIC comparators from the possible external additions.

**Note.** Additional functions that operate on internal comparators are described in section 16.4.

### 15.3.1 ConfigureInternalComparators

Prototype in C/C++

```
int eProDas_ConfigureInternalComparators(unsigned int DeviceIdx,
  unsigned int Mode, unsigned int InputSwitch, unsigned int Comp1Inv,
  unsigned int Comp2Inv);
```

Prototype in Delphi

```
function eProDas_ConfigureInternalComparators(DeviceIdx: LongWord; Mode: LongWord;
  InputSwitch: LongWord; Comp1Inv: LongWord; Comp2Inv: LongWord):integer;
```

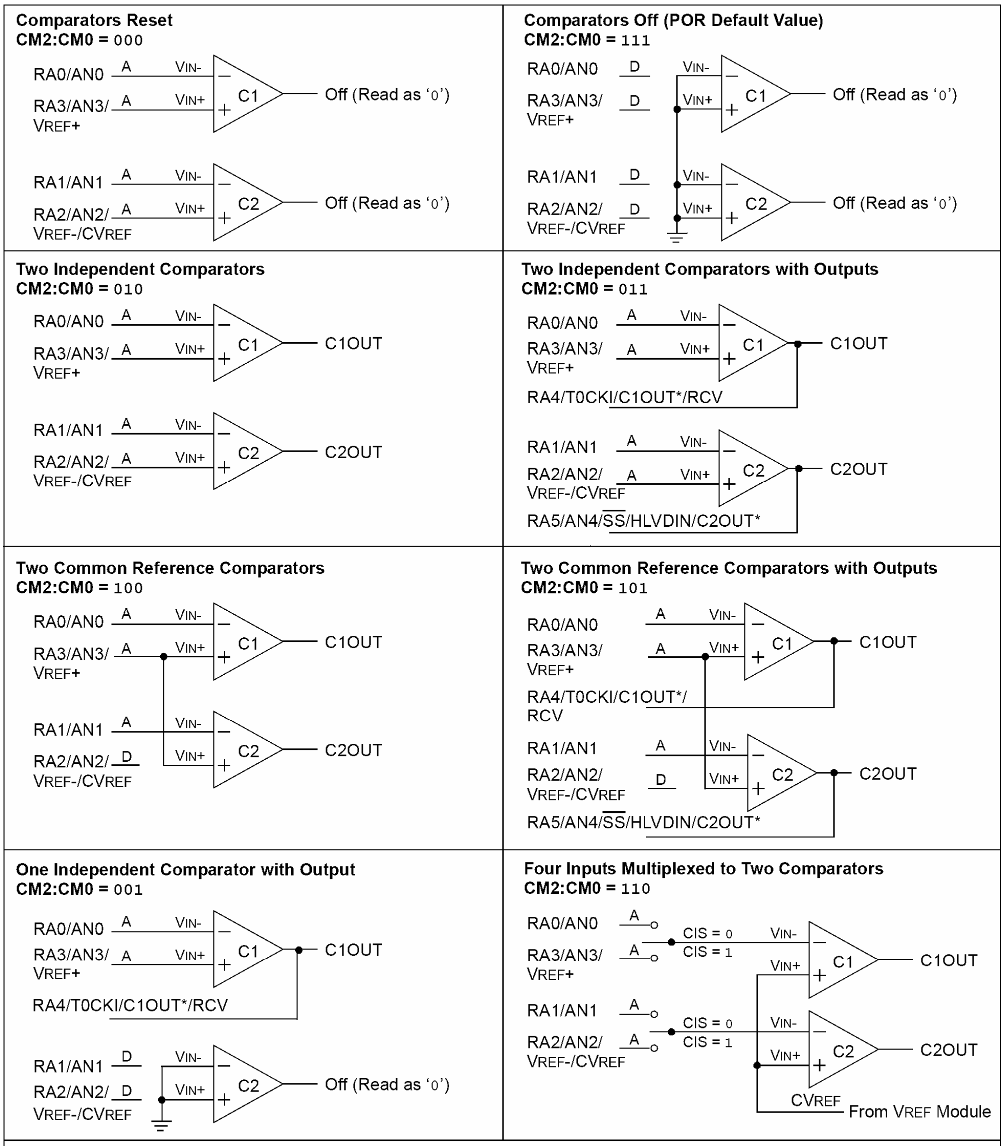
Prototype in VisualBasic

```
Function eProDas_ConfigureInternalComparators(ByVal DeviceIdx As UInteger,
  ByVal Mode As UInteger, ByVal InputSwitch As UInteger,
  ByVal Comp1Inv As   UInteger, ByVal Comp2Inv As UInteger) As Integer
```

With this function you tell the PIC how you want the comparators to behave. The `Mode` parameter selects one of the eight possible modes of operations that are illustrated in the Figure 37 (modes are displayed as binary values of CM2:CM0 bit fields). If `Mode` parameter is not between 0 and 7 an error `eProDas_Error_InvalidComparatorMode` will result.

For example, if selected `Mode` is 3 (CM2:CM0 = $011_2$), you can see from the figure that you obtain two independent comparators. The inputs of the first one are on pins RA0 and RA3, whereas its output is on pin RA4. Similarly, pins RA1 and RA2 are inputs of the second comparator and its output is connected to pin RA5.

**Note.** Although, in some of the listed modes comparator outputs are not available on external pins, you can always read their output values in software.



A = Analog Input, port reads zeros always D = Digital Input CIS (CMCON<3>) is the Comparator Input Switch
* Setting the TRISA<5:4> bits will disable the comparator outputs by configuring the pins as inputs.

**Figure 37: Modes of comparators module.**

If you select Mode 7 the comparator module turns off, whereas in mode 0 the module is operational but comparators always produce 0. If somebody knows of any use of this regime, please let us know.

Mode 6 is worth to mention separately, since here you can utilize a soon to be described voltage reference module to generate a reference voltage for comparison, which is especially suitable for building trigger circuits without additional external components.

With parameters `Comp1Inv` and `Comp2Inv` you independently select that one or both of the comparators invert their input(s) (the non-zero value of the respective parameter) if this makes more sense for your application. Generally, you may achieve the same effect by swapping the input pins, except that in mode 6 you cannot do that. There are other uses for these parameters, like software settable signal front for triggering.

Further, in mode 6 you can switch comparators' inputs between two sets of pins, which you do with the parameter `InputSwitch`. A zero value of this parameter selects pins RA0 and RA1 as inputs whereas a non-zero value assigns pins RA3 and RA2 for the job (see the CIS binary variable in the bottom right frame of the Figure 37).

**Note.** Comparator's output is a digital signal. If outputs are available on physical pins those pins must be configured as digital outputs or comparators will not output anything. For the sake of the safety, these pins are not automatically configured as digital outputs and you must do that by yourself with one of the `ConfigurePort(s)(Unsafe)` or `ConfigurePin` functions.

### 15.3.2 ReadInternalComparators

Prototype in C/C++

```
int eProDas_ReadInternalComparators(unsigned int DeviceIdx, unsigned char &Result);
```

Prototype in Delphi

```
function eProDas_ReadInternalComparators(DeviceIdx: LongWord;
  var Result: Byte):integer;
```

Prototype in VisualBasic

```
Function eProDas_ReadInternalComparators(ByVal DeviceIdx As UInteger,
  ByRef Result As Byte) As Integer
```

Regardless of the availability of comparators' outputs on physical pins you can always read their output value with this function. The bit 0 of parameter `Result` is the output of comparator 1 and bit 1 is the output of comparator 2.

## 15.4 Internal voltage reference

Besides comparators PIC also contains a voltage reference module, which may provide a suitable reference for comparison of the external voltage. The operation of this module is controlled by using only one function, since all you can do with it is to select the generated voltage.

### 15.4.1 ConfigureInternalReference

Prototype in C/C++

```
int eProDas_ConfigureInternalReference(unsigned int DeviceIdx,
  unsigned int Enabled, unsigned int Value, unsigned int Range,
  unsigned int SourceOnPin, unsigned int OutputOnPin);
```

Prototype in Delphi

```
function eProDas_ConfigureInternalReference(DeviceIdx: LongWord; Enabled: LongWord;
  Value: LongWord; Range: LongWord; SourceOnPin: LongWord;
  OutputOnPin: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_ConfigureInternalReference(ByVal DeviceIdx As UInteger,
  ByVal Enabled As UInteger, ByVal Value As UInteger, ByVal Range As UInteger,
  ByVal SourceOnPin As UInteger, ByVal OutputOnPin As UInteger) As Integer
```

Since one picture is worth thousands of words, the best way to grasp the idea of the workings of the module is to examine the Figure 38.

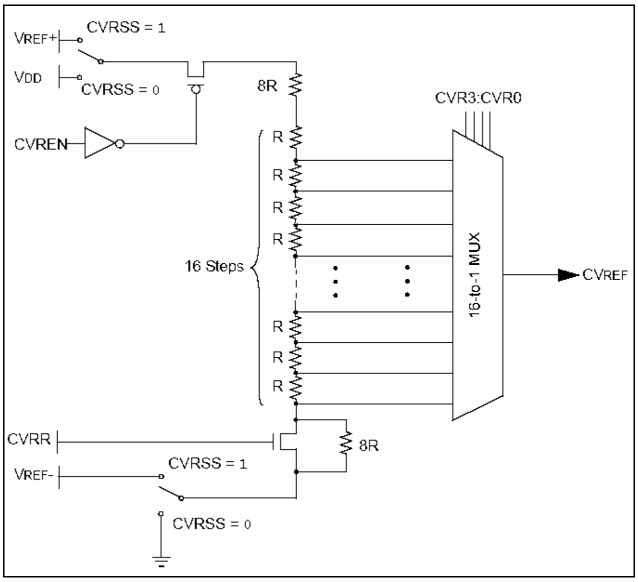


**Figure 38: Internals of the voltage reference.**

The voltage reference module is nothing more than the resistor ladder network with a selectable output. If parameter `Enabled` has a non-zero value the whole module is enabled (CVREN=1) and the power is supplied to the ladder network.

The `Value` parameter controls which of the 16 possible ladder connections gets connected to the output of the module CVREF. An error `eProDas_Error_InvalidReferenceValue` arises if this parameter is greater that 15.

With the `Range` parameter (CVRR in the picture) you control whether the ladder network is connected to the minus rail directly or through the additional resistor, by means of which you get two different voltage ranges and voltage steps. The formulas for calculating the output voltage are provided below.

A non-zero value of parameter `SourceOnPins` instructs that the ladder network is supplied by the reference voltage on pins RA2 and RA3; otherwise the network is connected to pins VDD and VSS. Note that you cannot control positive and negative voltage source separately as it is the case with the internal AD module.

Finally, a non-zero value of parameter `OutputOnPins` brings the CVREF voltage to the pin RA2. Please note, that there is no amplifier of this voltage on a chip so it is basically suitable only for connecting a voltmeter to it. Anything else requires external amplifier, like operational amplifier follower or some other high input impedance circuitry.

**Note.** The selection `OutputOnPins` is silently overridden if you also enable option `SourceOnPins`.

The nominal output voltage of the voltage reference module can be calculated by the following formulas:

- if `Range`=0: $\mathrm{CV_{REF}} = \left(\frac{\mathrm{V_{SRC}}}{4}\right) + \left(\frac{\mathtt{Value}}{32}\cdot \mathrm{V_{SRC}}\right) + \mathrm{V_{MINUS\_SRC}}$ ,
- if `Range`≠0: $\mathrm{CV_{REF}} = \left(\frac{\mathtt{Value}}{24}\cdot \mathrm{V_{SRC}}\right) + \mathrm{V_{MINUS\_SRC}}$ .

Voltage $V_{SRC}$ is the difference between the positive and negative resistor ladder supply rail and voltage $V_{MINUS\_SRC}$ is the potential of the negative supply rail.

For example, parameter `Value` equals 11, `Range` is 0 and `SourceOnPins` is also 0. Resistor ladder is supplied by voltages VSS and VDD. Therefore $V_{MINUS\_SRC}$ equals VSS = 0 V and $V_{SRC}$ equals (VDD-VSS) = 5 V. According to the first formula we calculate that the reference voltage equals slightly less than 3 V.

**Please note.** This module does not generate precise voltages. According to the datasheet the absolute accuracy is ½ LSB when Range=0 and only 1 LSB in the opposite case.

## 15.5 Pulse-width-modulation modules

There are two pulse-width-modulation (PWM) modules embedded in the PIC microcontroller. Module PWM2 enables generation of a simple single-output PWM signal whereas module PWM1 is a more sophisticated subsystem with capabilities of generating half- and full-bridge PWM signals in addition to PWM2 capabilities.

Both PWM modules share the same timer for generation of pulse period, therefore PWM period is the same for both modules and cannot be set independently. Contrary to this, duty cycles of PWM modules are completely independent, of course.

**Please note!** Some resources of module PWM2 are needed for generation of periodic actions and consequently these cannot be activated while module PWM2 is enabled.

### 15.5.1 PWM period

Period is determined by an 8-bit counter/timer (Timer2 according to PIC's datasheet) with settable period register PR and a prescaler of 1, 4 or 16. The input clock frequency of timer is 12 MHz. When Timer2 matches the period register the former is reset to zero upon the next (pre-scaled) clock pulse. Accordingly, the PWM period can be determined by the following formulas.

$$\text{Period}\left[\text{Clocks}\right] = (\text{PR} + 1) * \text{Prescaler}$$

$$\text{Period} = \frac{\text{Period}\left[\text{Clocks}\right]}{12\ \text{MHz}}$$

**Example 1.** The longest period is achieved by setting PR register to 255 and selecting prescaler of 16. According to the above formulas the period takes (255+1)*16 = 4096 clocks at 12 MHz, which is about 341.3 μs. The resulting PWM frequency is 2.93 kHz.

**Example 2.** For shorter periods the values of period register and/or prescaler needs to be reduced. By selecting PR value of 63 and prescaler of 1 the period decreases to 64 clocks at 12 MHz, which is about 5.3 μs and consequently the PWM frequency equals 187.5 kHz.

### 15.5.2 PWM duty cycle

Each PWM module integrates its own 10-bit duty cycle counter with input clock frequency of 48 MHz and the same prescaler as for the period counter. Again, user specifies the matching value, referred to as DR, upon which the counter is (immediately) reset. The duration of active PWM signal state and the related duty cycle can be calculated with the following formulas.

$$\text{Active State}\left[\text{Clocks}\right] = \text{DR} * \text{Prescaler}$$

$$\text{Active State} = \frac{\text{Active State}\left[\text{Clocks}\right]}{48\ \text{MHz}}$$

$$\text{Duty Cycle}\left[\%\right] = 100 \cdot \frac{\text{Active State}}{\text{Period}}$$

**Notes.**

1. If period register PR equals 255, duty cycle can be set with 10-bit resolution. With lower values of PR duty cycle resolution drops accordingly (9-bit when PR equals 127, 8-bit when PR is 63, etc.).

2. If duty cycle equals zero clocks (DR=0) PWM signal(s) stay(s) inactive at all times and there are no spikes on the PWM pin at the beginning of each PWM period. Therefore, zero duty cycle completely equals to the load being turned-off.

3. If active signal state is longer than the period the signal will remain active at all times.

Initial duty cycle of any PWM module is selected upon its configuration as described further on. In addition the following function is provided for changing duty cycle for each one or both PWM modules during their operation.

### 15.5.3 SetInternalPWMDutyCycle

Prototype in C/C++

```
int eProDas_SetInternalPWMDutyCycle(unsigned int DeviceIdx, int DutyCycle1, int
DutyCycle2);
```

Prototype in Delphi

```
function eProDas_SetInternalPWMDutyCycle(DeviceIdx: LongWord; DutyCycle1: Integer;
  DutyCycle2: Integer):integer;
```

Prototype in VisualBasic

```
Function eProDas_SetInternalPWMDutyCycle(ByVal DeviceIdx As UInteger,
  ByVal DutyCycle1 As Integer, ByVal DutyCycle2 As Integer) As Integer
```

**Note.** This function is intended to be used when PWM module(s) is(are) already configured and operational. Do not use this function to set duty cycle prior the configuration of the module(s) or else the settings will be overwritten upon module configuration.

Duty cycle for each PWM module can be specified independently by parameters `DutyCycle1` and `DutyCycle2`, each of which can hold a value from 0 to 1023. If duty cycle for a certain PWM module should remain unchanged the respective `DutyCycle` parameter should be set to –1.

For example, function call `SetInternalPWMDutyCycle(0, 512, -1)` sets duty cycle of PWM1 to 512 clocks and leaves duty cycle of PWM2 intact.

The error `eProDas_Error_InvalidDutyCycle` results if any of the `DutyCycle` parameters is outside of the range from –1 to 1023.

### 15.5.4 Module PWM2

Since module PWM2 is simpler to configure and work with, it is discussed first. Its configuration is done by two functions, where one of them is intended to fire up the module and the other shuts it down.

### 15.5.5 ConfigureInternalPWM2

Prototype in C/C++

```
int eProDas_ConfigureInternalPWM2(unsigned int DeviceIdx, int UnscaledPeriod,
  unsigned int Prescaler, unsigned int DutyCycle, unsigned int ConfigPin);
```

Prototype in Delphi

```
function eProDas_ConfigureInternalPWM2(DeviceIdx: LongWord;
  UnscaledPeriod: Integer; Prescaler: LongWord; DutyCycle: LongWord;
  ConfigPin: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_ConfigureInternalPWM2(ByVal DeviceIdx As UInteger,
  ByVal UnscaledPeriod As Integer, ByVal Prescaler As UInteger,
  ByVal DutyCycle As UInteger, ByVal ConfigPin As UInteger) As Integer
```

PWM period is specified by parameters `UnscaledPeriod` (PR register) and `Prescaler`, as described previously. When period is already configured (for example, if PWM1 is already operational) and you want to leave it intact, specify –1 for this parameter.

**Repeated note.** Both PWM modules have the same period. Whenever you change period of one PWM module, the change influences both PWM modules.

Function calls where `UnscaledPeriod` is greater than 255 or smaller than –1, lead to error `eProDas_Error_InvalidPeriod`. When `UnscaledPeriod` is not –1, `Prescaler` must have one of the values 1, 4 or 16, or else error `eProDas_Error_InvalidPrescaler` will result.

When `UnscaledPeriod` is –1, PWM period must have already been configured (Timer2 must be active, for example by configuring the other PWM module prior to calling this function) or else error `eProDas_Error_TimerDisabled` will result.

In order for PWM signal to be generated, PWM pin must be configured as digital output. This can be done automatically if you specify a non-zero value for parameter `ConfigPin`. Alternatively, you can take care of this by yourself.

PWM pin for module PWM2 can be either RC1 or RB3. This is not settable in a usual way, but it is pre-programmed into special configuration bits of PIC microcontroller. Use the function `SetInternalPWM2OutputPin` (section 15.5.7) to make the selection. The actual pin configuration is reported by function `GetConfiguration` (section 22.1) and by Console demo application.

**Repeated note!** Some resource of module PWM2 are needed for generation of periodic actions and consequently these cannot be activated while module PWM2 is enabled.

### 15.5.6 TurnOffInternalPWM2

Prototype in C/C++

```
int eProDas_TurnOffInternalPWM2(unsigned int DeviceIdx, unsigned int ConfigPin);
```

Prototype in Delphi

```
function eProDas_TurnOffInternalPWM2(DeviceIdx: LongWord;
  ConfigPin: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_TurnOffInternalPWM2(ByVal DeviceIdx As UInteger,
  ByVal ConfigPin As UInteger) As Integer
```

Use this function to shuts down the module PWM2. Do not just specify zero duty cycle to stop PWM signal, since this way the module is still operational, it consumes power and prevents PWM output pin to take role of a general-purpose digital I/O pin.

Non-zero value of the parameter `ConfigPin` configures PWM pin as digital input in the process. Alternatively, pin configuration is kept untouched.

### 15.5.7 SetInternalPWM2OutputPin

Prototype in C/C++

```
int eProDas_SetInternalPWM2OutputPin(unsigned int DeviceIdx,
  unsigned int OutputPin);
```

Prototype in Delphi

```
function eProDas_SetInternalPWM2OutputPin(DeviceIdx: LongWord;
  OutputPin: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_SetInternalPWM2OutputPin(ByVal DeviceIdx As UInteger,
  ByVal OutputPin As UInteger) As Integer
```

With this function your application can select whether the output signal of PWM2 module is generated on pin RB3 or RC1. In the former case the value of parameter `OutputPin` must equal zero, whereas in the latter one it must differ from zero.

**Important.** The configuration of PWM2 output pin is not written into an ordinary PIC register. Instead, it is stored into a special non-volatile configuration bit, which means that the setting is preserved even when the device is turned off.

Unfortunately, all types of non-volatile memory are subject to some limited write/erase endurance. Specifically, if you change the configuration of PWM2 output pin a couple of 10,000 times you will eventually destroy the PIC. In order to make this scenario as unlikely as possible the function `SetInternalPWM2OutputPin` first reads the old value of the configuration and initiates the actual write step only if the new requested setting differs from the old one. This way it is safe to specify PWM2 output pin at the beginning of you application, which will not be written to non-volatile storage each time the application is executed. However, if you make a program loop where you constantly alter the choice of output pin, you will destroy your chip in a short time.

**Note 1.** With this function you may change PWM2 output pin on the fly, i.e. even during the time when PWM2 module is active. However, it is expected that you will use this function primarily at the beginning of experiments. Therefore, this function does not take care of automatically configuring new output pin as digital output if PWM2 module is operational at the time of pin switch. The next call of function `ConfigureInternalPWM2` properly handles the issue. Of course, you may always configure the direction of pin by yourself prior or after changing the PWM2 output pin.

**Note 2.** The same pin that is in charge of outputting the PWM2 signal is also responsible for a completely unrelated task. Namely, when performing high-resolution measurements of delay (section 16.7.1) the very same pin plays the role of digital input.

### 15.5.8 Module PWM1

Module PWM1 is significantly more capable than module PWM2. Consequently, the user is required to specify noticeably more settings when configuring this module in comparison to configuration of module PWM2. Also, additional functionality requires more functions to be provided for controlling the module's activity. Fortunately, the thorough description follows.

### 15.5.9 ConfigureInternalPWM1

Prototype in C/C++

```
int eProDas_ConfigureInternalPWM1(unsigned int DeviceIdx, int UnscaledPeriod,
  unsigned int Prescaler, unsigned int DutyCycle, unsigned int Mode,
  unsigned int ActiveState, unsigned int Delay, unsigned int AutoShutDownSource,
  unsigned int AutoShutDownPinState, unsigned int AutoRestart);
```

Prototype in Delphi

```
function eProDas_ConfigureInternalPWM1(DeviceIdx: LongWord;
  UnscaledPeriod: Integer; Prescaler: LongWord; DutyCycle: LongWord;
  Mode: LongWord; ActiveState: LongWord; Delay: LongWord;
  AutoShutDownSource: LongWord; AutoShutDownPinState: LongWord;
  AutoRestart: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_ConfigureInternalPWM1(ByVal DeviceIdx As UInteger,
  ByVal UnscaledPeriod As Integer, ByVal Prescaler As UInteger,
  ByVal DutyCycle As UInteger, ByVal Mode As UInteger,
  ByVal ActiveState As UInteger, ByVal Delay As UInteger,
  ByVal AutoShutDownSource As UInteger, ByVal AutoShutDownPinState As UInteger,
  ByVal AutoRestart As UInteger) As Integer
```

Parameters `UnscaledPeriod`, `Prescaler` and `DutyCycle` have precisely the same meaning as before. Please, see their descriptions in section 15.5.5.

Parameter `Mode` specifies one of the four possible modes of PWM1 operation:

0. single output on pin P1A (RC2); equivalent to operation of module PWM2,
1. half-bridge mode with two modulated pins P1A (RC2) and P1B (RD5),
2. full-bridge mode in forward direction, with four pins P1A (RC2) and P1B (RD5), P1C (RD6) and P1D (RD7),
3. full-bridge mode in reverse direction and with the same set of pins as in forward mode.

Other values of parameter `Mode` result in error `eProDas_Error_InvalidMode`.

The four modes are summarized in the Figure 39 and the last three of them are described further on.

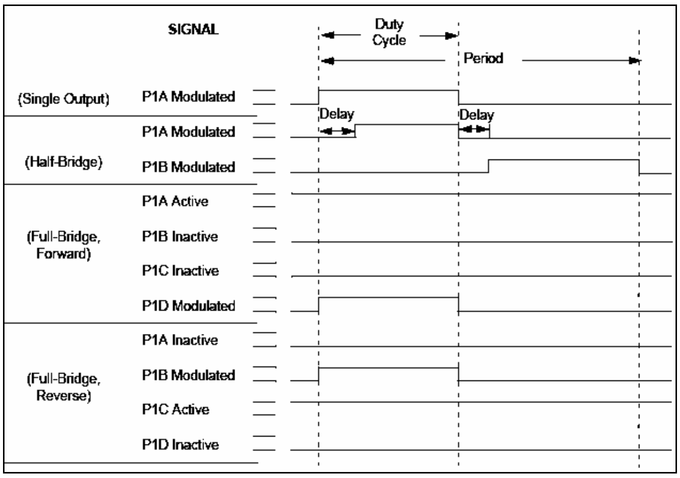


**Figure 39: Summary of PWM1 modes of operation.**

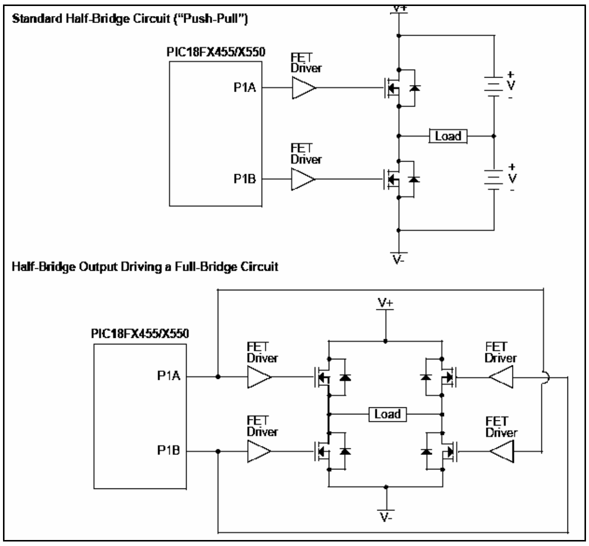


**Figure 40: Half-bridge example applications.**

Two example applications of half-bridge mode of operation are presented in the Figure 40. Here one of the PWM pins is active for the duration of duty cycle interval and the other one for the remaining of the PWM period. This way the load can be excited with positive as well as negative average voltages if bipolar power supply is utilized.

There is one serious problem with this configuration, however. Observing the upper (or lower, for that matter) schematic in the Figure 40, we can realize the following. If, for example, signal P1A opens the upper FET switch before the lower one is fully turned-off, a short-circuit condition arises with the consequences of increased dissipation (excess energy consumption and transistor heating) at best and with the destruction of FET switches or power supply (highly unlikely) at worst.

In order to prevent this scenario a short delay needs to be inserted after deactivation of one PWM signal and before activation of the other. The length of this delay is specified by the parameter `Delay`, which determines delay duration in number of clock cycles at 12 MHz. The value of this parameter must be between 0 and 127 or error `eProDas_Error_InvalidDelay` will result. For other modes of operation this parameter is ignored.

Example application of full-bridge mode of operation is presented in the Figure 41. The beauty of this approach to PWM control is that load can be supplied by both voltage polarities although only single polarity power supply is used. The drawback is that load cannot be referenced to any of the supply rails.

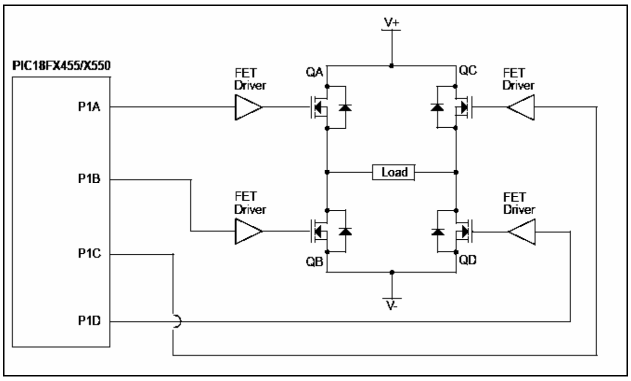


**Figure 41: Example application for full-bridge operation.**

The actual polarity of signal voltage (the direction) is determined by the forward or backward variant of full-bridge mode of operation. For determination and instant change of the direction two functions are provided and described in the next sections.

By parameter `ActiveState` the user specifies whether PWM pins are active (i.e. the associated switches are conducting) in low or high state, according to the following list:

0. all PWM pins are active high,
1. pins P1A (RC2) and P1C (RD6) are active high, pins P1B (RD5) and P1D (RD7) are active low,
2. pins P1A (RC2) and P1C (RD6) are active low, pins P1B (RD5) and P1D (RD7) are active high,
3. all PWM pins are active low.

An error `eProDas_Error_InvalidActiveState` occurs if parameter `ActiveState` does not hold one of the above specified values.

Module PWM1 possesses the auto shutdown feature, by means of which it automatically stops exciting the load (generation of PWM signals asynchronously ceases) whenever certain user-specific event(s) occur. This way the user can arrange PWM system in such a way that e.g. load, power-supply or switches cannot be overloaded.

There are three auto shutdown sources that can be arbitrarily activated by specifying the proper value of parameter `AutoShutDownSource` according to the following list:

0. auto shutdown is disabled,
1. internal comparator 1 output,
2. internal comparator 2 output,
3. either comparator 1 or 2 output,
4. state on pin RB0 (FLT0 function according to datasheet),
5. state on pin RB0 or internal comparator 1,
6. state on pin RB0 or internal comparator 2,
7. any of the above sources.

If parameter `AutoShutDownSource` does not hold one of the above specified values an error `eProDas_Error_InvalidAutoShutDownSource` results.

Parameter `AutoShutDownPinState` specifies pin states during the auto shutdown condition, according to the following list:

0. all pins are driven low,
1. pins P1B (RD5) and P1D (RD7) are driven high, the other two are driven low,
2. pins P1B (RD5) and P1D (RD7) are in high impedance, the other two are driven low,
3. the same as 2,
4. pins P1A (RC2) and P1C (RD6) are driven high, the other two are driven low,
5. all pins are driven high,
6. all pins are in high impedance state,
7. the same as 6.

If parameter `AutoShutDownPinState` does not hold proper value and `AutoShutDownSource` holds a non-zero value an error `eProDas_Error_InvalidAutoShutDownPinState` results.

Finally, a non-zero value of parameter `AutoRestart` specifies that PWM module exits auto shutdown state automatically when the shutdown condition ceases to exist. Otherwise, the user needs to call function `eProDas_ClearInternalPWM1AutoShutDownState` explicitly to put PWM module out of the shutdown state.

### 15.5.10 ClearInternalPWM1AutoShutDownState

Prototype in C/C++
```
int eProDas_ClearInternalPWM1AutoShutDownState(unsigned int DeviceIdx);
```

Prototype in Delphi
```
function eProDas_ClearInternalPWM1AutoShutDownState(DeviceIdx: LongWord):integer;
```

Prototype in VisualBasic
```
Function eProDas_ClearInternalPWM1AutoShutDownState(ByVal DeviceIdx As UInteger)
  As Integer
```

Use this function to put PWM1 module out of shutdown state. If the module is configured to automatically exit this state or if the condition to enter it has not arisen, this function has no effect.

### 15.5.11 GetInternalPWM1AutoShutDownState

Prototype in C/C++
```
int eProDas_GetInternalPWM1AutoShutDownState(unsigned int DeviceIdx,
  unsigned int &State);
```

Prototype in Delphi
```
function eProDas_GetInternalPWM1AutoShutDownState(DeviceIdx: LongWord;
  var State: LongWord):integer;
```

Prototype in VisualBasic
```
Function eProDas_GetInternalPWM1AutoShutDownState(ByVal DeviceIdx As UInteger,
  ByRef State As UInteger) As Integer
```

Use this function to check whether module PWM1 is currently in the shutdown state (a non-zero value of variable `State`) or not (zero returned value in the variable `State`).

### 15.5.12 ChangeInternalPWM1FullBridgeDirection

Prototype in C/C++

```
int eProDas_ChangeInternalPWM1FullBridgeDirection(unsigned int DeviceIdx);
```

Prototype in Delphi

```
function eProDas_ChangeInternalPWM1FullBridgeDirection(
  DeviceIdx: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_ChangeInternalPWM1FullBridgeDirection(ByVal DeviceIdx As UInteger)
  As Integer
```

Use this function to instantly switch between forward and reverse mode operation when full-bridge PWM regime is activated.

### 15.5.13 GetInternalPWM1FullBridgeDirection

Prototype in C/C++

```
int eProDas_GetInternalPWM1FullBridgeDirection(unsigned int DeviceIdx,
  unsigned int &Direction);
```

Prototype in Delphi

```
function eProDas_GetInternalPWM1FullBridgeDirection(DeviceIdx: LongWord;
  var Direction: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_GetInternalPWM1FullBridgeDirection(ByVal DeviceIdx As UInteger,
  ByRef Direction As UInteger) As Integer
```

Use this function to establish whether forward (zero returned value in the variable `Direction`) or reverse (a non-zero value of the variable `Direction`) mode of full-bridge PWM regime is currently active.

### 15.5.14 TurnOffInternalPWM1

Prototype in C/C++

```
int eProDas_TurnOffInternalPWM1(unsigned int DeviceIdx, unsigned int ConfigPin);
```

Prototype in Delphi

```
function eProDas_TurnOffInternalPWM1(DeviceIdx: LongWord;
  ConfigPin: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_TurnOffInternalPWM1(ByVal DeviceIdx As UInteger,
  ByVal ConfigPin As UInteger) As Integer
```

This function works in a precisely the same manner as the function `TurnOffInternalPWM2`, except that it acts on module PWM1 instead on the module PWM2.

## 15.6 SPI interface

On hardware level PIC supports communication with digital periphery through the SPI (Serial Peripheral Interface) protocol/specification. Supporting this type of data exchange is essential for an extensible data acquisition system like eProDas, since many excellent AD and DA converters as well as other chips rely on this communication protocol. The increased popularity of serial interfaces all over the computer and digital industry should come as no surprise, since wiring 3 or four wires is much easier, practical as well as space and labour saving that wiring, say, 25 of them.

When connecting the external SPI enabled periphery to the PIC, make connections according to the following scheme (see also the Figure 2 on the page 21). Pin SDO (serial data out) is on PIC's pin RC7/SDO, pin SDI (serial data in) is on pin RB0/SDI and serial clock SCK is on pin RB1/SCK. If PIC works as a slave with pin Slave Select (SS) enabled (see further on), this pin is located at RA5/SS.

**Note.** Pin RC7/SDO of SPI module is shared with USART module, where the role of the pin is RC7/RX. Such selection is extremely unfortunate since the pin must be configured as digital output to fulfil its SPI role, but it must be digital input for proper working of USART module. If you use both of these modules in your application, please read section 16.6, which explains how eProDas can help you living with this idiosyncrasy.

PIC supports master as well as slave mode of SPI communication. In the former case PIC generates serial clock and initiates data transfers whereas in the opposite case some other master takes on this role. Consequently, there exist separate eProDas functions for SPI module configuration depending on its preferred master or slave role.

### 15.6.1 SetInternalSPI_MasterMode

Prototype in C/C++

```
int eProDas_SetInternalSPI_MasterMode(unsigned int DeviceIdx,
  unsigned int IN_SampleAtTheEnd, unsigned int ClockIdleStateHigh,
  unsigned int ClockSpeed, unsigned int ClockPeriod, unsigned int ClockPrescaler,
  unsigned int CorrectFirstBit, unsigned int InputOutput,
  unsigned int EnableOutput);
```

Prototype in Delphi

```
function eProDas_SetInternalSPI_MasterMode(DeviceIdx: LongWord;
  IN_SampleAtTheEnd: LongWord; ClockIdleStateHigh: LongWord; ClockSpeed: LongWord;
  ClockPeriod: LongWord; ClockPrescaler: LongWord;
  CorrectFirstBit: LongWord; EnableInput: LongWord;
  EnableOutput: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_SetInternalSPI_MasterMode(ByVal DeviceIdx As UInteger,
  ByVal IN_SampleAtTheEnd As UInteger, ByVal ClockIdleStateHigh As UInteger,
  ByVal ClockSpeed As UInteger, ByVal ClockPeriod As UInteger,
  ByVal ClockPrescaler As UInteger, ByVal CorrectFirstBit As UInteger,
  ByVal EnableInput As UInteger, ByVal EnableOutput As UInteger) As Integer
```

With non-zero value of parameter `IN_SampleAtTheEnd` you instruct PIC to sample input data at clock transition that finishes the interval in which the asserted bit is valid; otherwise the signal is sampled in the middle of the interval.

A non-zero value of parameter `ClockIdleStateHigh` selects that clock is held in high state (logic 1) during its inactive (idle) periods; otherwise the clock's idle state is low (logic 0).

With parameter `ClockSpeed` you specify the desired frequency of serial clock during data transfer. If this parameter equals 0, 1 or 2 them 12 MHz, 3 MHz or 750 kHz clock, respectively, is used for the task; i.e. there is a maximal data rate of 12 Mbps, 3 Mbps or 750 kbps in each direction, respectively.

There is also a possibility of fine grained clock control for which the two parameters `ClockPeriod` and `ClockPrescaler` are used if you specify the value of 3 for parameter the `ClockSpeed` (otherwise these two parameters are ignored). When you exploit the fine grained clock control, the period of serial clock is determined by the same 8-bit counter/timer Timer2 that controls PWM period as described in the section 15.5.1 (parameters `ClockPeriod` and `ClockPrescaler` equal PR register and timer prescaler, respectively, and their meaning is explained in the PWM section). Note: SPI clock is obtained by dividing Timer2 output by 2, i.e. SPI clock period is twice as long as the period of equally configured PWM module would be.

You can configure SPI clock as described in the previous paragraph only if both PWM modules are turned off, otherwise a conflict in the form of `eProDas_Error_ResourceInUse` error will result. Alternatively, you can arrange things in such a way that SPI clock can be controlled by the same Timer2 settings as are needed for synthesizing the PWM period (SPI clock is still one half of the PWM clock). In this case you select 4 for the parameter the `ClockSpeed` and eProDas system will ignore parameters `ClockPeriod` and `ClockPrescaler` but it will instead use current Timer2 configuration.

**Note 1.** Timer2 must be already configured prior calling this function (i.e. configure PWM module first) or the error `eProDas_Error_TimerDisabled` will result.

**Note 2.** When using Timer2 for SPI time base PIC reveals a certain flaw. Namely, SPI transfer starts asynchronously with the Timer2 period, which means that the first bit can be transferred in a much shorter time than expected (depending on the value of Timer2 counter at the moment of initiation of the transfer). If external periphery cannot cope with the resulting shorter bit period the transfer will not be done successfully. eProDas can compensate for this phenomena in firmware by manipulating Timer2 counter value, however this disrupts the current period of PWM module(s) if it(they) is(are) active. Therefore, eProDas cannot decide for you whether the compensation should be done since the choice depends on the circumstances. For that matter the functions for configuring SPI module as a master possess a parameter `CorrectFirstBit`, through which you give eProDas a hint about whether the length of the first bit should be corrected (non-zero value) or not.

With non-zero value of the parameter `EnableInput` you indicate that you intend to receive data from external periphery and for that matter PIC’s pin RB0/SDI gets automatically configured as digital input. Alternatively, the zero value of the parameter leaves the configuration of pin RB0 untouched. By not receiving anything from SPI bus you gain one digital I/O pin, which can be assigned to some other task.

Bear in mind that the parameter `EnableInput` only serves to prevent altering configuration of pin RB0/SDI. If you call some function that does SPI input transfer (described further on), some data will be delivered to you anyway, according to the state of pin RB0. For example, if pin RB0 is configured as digital output then the data that you write to port’s B bit 0 will dictate the SPI received values.

With non-zero value of the parameter `EnableOutput` you indicate that you intend to send data from PIC to external periphery and for that matter PIC’s pin RC7/SD0 gets automatically configured as digital output. Alternatively, the zero value of the parameter configures the respective pin as digital input by means of which you gain one more digital input pin (the pin cannot remain digital output since this way it would be automatically driven by SPI engine during transfers). Of course, now you can only receive data over the SPI bus but you cannot send anything to the outside SPI world.

Bear in mind that the parameter EnableOutput only serves to configure pin RC7/SDO according to your selection. If you call functions for SPI output transfer (described further on) you must still provide some output data, which will be put into the well without the bottom by the RC7's hardware configuration.

**Note.** When you do not need to send any data over SPI bus there may be a good reason to disable SPI output. Namely, if your application utilizes both the SPI and the USART module (section 15.7), then generally you need to take great pain of constantly (re)configuring pin RC7/RX/SDO either as input or output depending on to which of the two modules it is associated at a time (in this case you are kindly urged to read section 16.6). However, if you can live without SPI output transfer, pin RC7 can be devoted to USART module exclusively and your life becomes much more gay and cheerful.

### 15.6.2 SetInternalSPI_MasterMode_CS

Prototype in C/C++

```
int eProDas_SetInternalSPI_MasterMode_CS(unsigned int DeviceIdx,
  unsigned int IN_SampleAtTheEnd, unsigned int ClockIdleStateHigh,
  unsigned int ClockSpeed, unsigned int ClockPeriod, unsigned int ClockPrescaler,
  unsigned int ChipSelectPort, unsigned int ChipSelectBit,
  unsigned int CorrectFirstBit, unsigned int EnableInput,
  unsigned int EnableOutput);
```

Prototype in Delphi

```
function eProDas_SetInternalSPI_MasterMode_CS(DeviceIdx: LongWord;
  IN_SampleAtTheEnd: LongWord; ClockIdleStateHigh: LongWord; ClockSpeed: LongWord;
  ClockPeriod: LongWord; ClockPrescaler: LongWord; ChipSelectPort: LongWord;
  ChipSelectBit: LongWord; ChipSelectState: LongWord;
  CorrectFirstBit: LongWord; EnableInput: LongWord;
  EnableOutput: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_SetInternalSPI_MasterMode_CS(ByVal DeviceIdx As UInteger,
  ByVal IN_SampleAtTheEnd As UInteger, ByVal ClockIdleStateHigh As UInteger,
  ByVal ClockSpeed As UInteger, ByVal ClockPeriod As UInteger,
  ByVal ClockPrescaler As UInteger, ByVal ChipSelectPort As UInteger,
  ByVal ChipSelectBit As UInteger, ByVal ChipSelectState As UInteger,
  ByVal CorrectFirstBit As UInteger, ByVal EnableInput As UInteger,
  ByVal EnableOutput As UInteger) As Integer
```

Generally, your SPI chip needs a proper handling of *ChipSelect* or semantically equal pin with different name. In the majority of cases the pin must be held in state logic 1 prior initiating any communication with the chip, changed to logic 0 just prior starting the communication and returned to logic 1 afterwards. In simple cases where no address decoding logic is implemented, the described sequence can be implemented by hooking *ChipSelect(like)* pin to a spare PIC's pin and manipulating port's latch register properly by using functions from the section 15.1. However, eProDas functions can help you a little more with this tedious and boring task.

The function SetInternalSPI_MasterMode_CS works in the same way as the function SetInternalSPI_MasterMode does, except that in addition it also configures one PIC's pin as digital output and puts it into initial logic state of your choice; always select the state that renders the chip or module inactive (usually logic 1). It is not an error if the pin is already configured as digital output; in this case the function still makes sure that the pin state is set according to your selection.

The parameter ChipSelectPort specifies the port (Table 4) to which pin *Chip Select* of the periphery in question is connected to. If you specify the value of 1 000 or more for this parameter, the function skips pin configuration and pretends that its name is SetInternalSPI_MasterMode without _CS suffix.

The parameter ChipSelectBit specifies the actual bit or pin index (from 0 to 7). For example, if chip select is connected to pin RB4 of the PIC, specify the value of 1 for ChipSelectPort and the value of 4 for ChipSelectBit.

If more than one SPI chip is connected to PIC then you will probably have to manually configure *ChipSelect* handling pins by using functions in the section 15.1 or in complicated cases by using address decoding logic (see section 16.2).

### 15.6.3 SetInternalSPI_SlaveMode

Prototype in C/C++

```
int eProDas_SetInternalSPI_SlaveMode(unsigned int DeviceIdx,
  unsigned int TransmitOnActiveToIdle, unsigned int SlaveSelectEnable,
  unsigned int ClockIdleStateHigh, unsigned int EnableOutput);
```

Prototype in Delphi

```
function eProDas_SetInternalSPI_SlaveMode(DeviceIdx: LongWord;
  TransmitOnActiveToIdle: LongWord; SlaveSelectEnable: LongWord;
  ClockIdleStateHigh: LongWord; EnableOutput: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_SetInternalSPI_SlaveMode(ByVal DeviceIdx As UInteger,
  ByVal TransmitOnActiveToIdle As UInteger, ByVal SlaveSelectEnable As UInteger,
  ByVal ClockIdleStateHigh As UInteger, ByVal EnableOutput As UInteger) As Integer
```

In this mode you do not configure SPI clock since this signal is under the control of some other master circuit, but there are some other configuration parameters that you need to set properly to comply with the master's mode of operation.

A non-zero value of TransmitOnActiveToIdle instructs PIC to output the next bit of data on clock transition from active to idle state; otherwise the bit transition occurs on clock transition from idle to active state.

With non-zero value of parameter SlaveSelectEnable you instruct the PIC to respond to serial transfers only when pin RA5/SS is in low state. This is used (of course) when there are more slaves on a serial bus and by means of pin SS the master somehow selects to which chip it intends to talk.

A non-zero value of parameter ClockIdleStateHigh informs the PIC that clock is held in high state (logic 1) during its inactive (idle) periods; otherwise the idle state is low (logic 0).

For the description of parameters EnableInput and EnableOutput, please see the description of function SetInternalSPI_MasterMode (section 15.6.1).

**Note.** So far, eProDas SPI transfer functions only support master mode of operation, therefore this functions is of **no practical value**. Please, let us know if you need SPI slave transfer and we will implement it ASAP.

### 15.6.4 TurnOffInternalSPI

Prototype in C/C++

```
int eProDas_TurnOffInternalSPI(unsigned int DeviceIdx,
  unsigned int ShutDownTimer2);
```

Prototype in Delphi

```
function eProDas_TurnOffInternalSPI(DeviceIdx: LongWord;
  ShutDownTimer2: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_TurnOffInternalSPI(ByVal DeviceIdx As UInteger,
  ByVal ShutDownTimer2 As UInteger) As Integer
```

With this function you disable SPI interface and restore the usual digital I/O and other functionality of pins that are dedicated to SPI. All pins that were configured as digital outputs (serial clock, serial data out) by the function SetInternalSPI_*XXX*Mode are reconfigured as digital inputs. Potential digital outputs that were configured as inputs (serial data in) are not reconfigured back as outputs automatically. Also, *ChipSelect* pin configuration of the function SetInternalSPI_MasterMode_CS is left untouched.

Timer2 is shut down if (and only if) it is used for serial clock synthesis (you selected the value of 3 or 4 for the parameter ClockSpeed upon calling the function SetInternalSPI_MasterMode). If this is not desirable (if PWM is still using Timer2 at this time) specify zero value for parameter ShutDownTimer2 and configuration of Timer2 will be left untouched.

**Note.** You need to call this function if SPI module is currently active and you want to reconfigure it (different clock speed or other settings) before recalling the function SetInternalSPI_*XXX*Mode.

### 15.6.5 Transferring data over SPI bus

This section presents a basic subset of eProDas functions for transferring single and multi-byte data packets over SPI bus. Although this seems to be everything that you will ever need, the praxis is not so simple. There exist many different demands, which SPI periphery imposes on an eProDas system. There is a need to handle *ChipSelect* pin properly, trigger *LoadStrobe* pin at the end of the transfer, wait for *DataReady* signal before exchanging data, to name a few. For that matter, eProDas API defines many SPI related functions that are described in section 16.5 in addition to the ones that are presented here and we strongly suggest you to learn about them before you start coding your SPI application. Further, for the most demanding cases there exists a Transfer backbone (section 23.2), which constellates SPI bus cycles according to your description by means of which you can implement next to any SPI communication specification.

### 15.6.5.1 MasterSPI_TransferByte group of functions

Prototypes in C/C++

```
int eProDas_MasterSPI_TransferByte(unsigned int DeviceIdx, unsigned char WriteByte,
  unsigned char &ReadByte);

int eProDas_MasterSPI_TransferByteOUT(unsigned int DeviceIdx,
  unsigned char WriteByte);

int eProDas_MasterSPI_TransferByteIN(unsigned int DeviceIdx,
  unsigned char &ReadByte);
```

Prototypes in Delphi

```
function eProDas_MasterSPI_TransferByte(DeviceIdx: LongWord; WriteByte: Byte;
  var ReadByte: Byte):integer;

function eProDas_MasterSPI_TransferByteOUT(DeviceIdx: LongWord; WriteByte: Byte;
  ):integer;

function eProDas_MasterSPI_TransferByteIN(DeviceIdx: LongWord; var ReadByte: Byte
  ):integer;
```

Prototypes in VisualBasic

```
Function eProDas_MasterSPI_TransferByte(ByVal DeviceIdx As UInteger,
  ByVal WriteByte As Byte, ByRef ReadByte As Byte) As Integer

Function eProDas_MasterSPI_TransferByteOUT(ByVal DeviceIdx As UInteger,
  ByVal WriteByte As Byte) As Integer

Function eProDas_MasterSPI_TransferByteIN(ByVal DeviceIdx As UInteger,
  ByRef ReadByte As Byte) As Integer
```

These functions work when PIC is configured as SPI master and they transfer one byte of data, the value of the parameter `WriteByte`, over SPI bus to the connected slave peripheral circuitry. As it is specified by SPI protocol, PIC always simultaneously receives one byte of data from the periphery; this data is returned in the parameter `ReadByte`. According to SPI standard it is not possible to only send or receive data, but you can always ignore the meaningless part of the transfer, that is why there exist three functions within this group.

Use the function `MasterSPI_TransferByte` when transfer into both directions is meaningful to your application. The function `MasterSPI_TransferByteOUT` only sends the byte from PIC to periphery and discards the received byte by means of which you save a bit of a time since meaningless value is not transferred over the USB bus. Similarly, use the function `MasterSPI_TransferByteIN` when you only want to receive the byte from periphery.

### 15.6.5.2 MasterSPI_TransferArray group of functions

Prototypes in C/C++

```
eProDasDLL_API int eProDas_MasterSPI_TransferArray(unsigned int DeviceIdx,
  unsigned char *WriteArray, unsigned char *ReadArray, unsigned int Length);

int eProDas_MasterSPI_TransferArrayOUT(unsigned int DeviceIdx,
  unsigned char *WriteArray, unsigned int Length);

int eProDas_MasterSPI_TransferArrayIN(unsigned int DeviceIdx,
  unsigned char *ReadArray, unsigned int Length);
```

Prototypes in Delphi

```
function eProDas_MasterSPI_TransferArray(DeviceIdx: LongWord; WriteArray: PByte;
  ReadArray: PByte; Length: LongWord):integer;

function eProDas_MasterSPI_TransferArrayOUT(DeviceIdx: LongWord; WriteArray: PByte;
  Length: LongWord):integer;

function eProDas_MasterSPI_TransferArrayIN(DeviceIdx: LongWord; ReadArray: PByte;
  Length: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_MasterSPI_TransferArray(ByVal DeviceIdx As UInteger,
  ByRef WriteArray As Byte, ByRef ReadArray As Byte, ByVal Length As UInteger)
  As Integer

Function eProDas_MasterSPI_TransferArrayOUT(ByVal DeviceIdx As UInteger,
  ByRef WriteArray As Byte, ByVal Length As UInteger) As Integer

Function eProDas_MasterSPI_TransferArrayIN(ByVal DeviceIdx As UInteger,
  ByRef ReadArray As Byte, ByVal Length As UInteger) As Integer
```

These functions work in precisely the same way as the group `MasterSPI_TransferByte` does, except that they can transfer up to 39 bytes of data in each direction. The data to be transferred to the periphery is contained in the byte array to which the parameter `WriteArray` points to. The received data gets written to the array that is pointed to by the parameter `ReadArray`. The requested transfer length is specified by the parameter `Length`.

Since SPI supports only simultaneous bidirectional transfers, one `Length` specifies both array sizes. Both arrays are fully buffered internally so the functions work correctly even if you specify the same pointer for both of them, which is useful if you intend to discard write array after the transfer is done and you do not want to allocate two separate buffers for the data.

## 15.6.6 Reconfiguration of SPI clock

Generally, you want to connect different chips to the PIC. Usually, these chips require different SPI clock's frequency and other settings to operate properly. The following function enables you to reconfigure SPI clock to suit different demands imposed by your periphery; parameters are equal to the corresponding parameters of the function `SetInternalSPI_MasterMode` (section 15.6.1).

#### 15.6.6.1 Configure_SPI_Clock

Prototypes in C/C++

```
int eProDas_Configure_SPI_Clock(unsigned int DeviceIdx,
  unsigned int IN_SampleAtTheEnd, unsigned int ClockIdleStateHigh,
  unsigned int ClockSpeed, unsigned int ClockPeriod, unsigned int ClockPrescaler,
  unsigned int CorrectFirstBit);
```

Prototypes in Delphi

```
function eProDas_Configure_SPI_Clock(DeviceIdx: LongWord;
  IN_SampleAtTheEnd: LongWord; ClockIdleStateHigh: LongWord; ClockSpeed: LongWord;
  ClockPeriod: LongWord; ClockPrescaler: LongWord;
  CorrectFirstBit: LongWord):integer;
```

Prototypes in VisualBasic

```
eProDas_Configure_SPI_Clock
```

## 15.7 Universal serial receiver transmitter (USART)

Besides synchronous serial transfers like SPI transfer is, PIC also supports asynchronous serial transfers like RS-232. The name of PIC’s module for the task is USART (universal serial receiver transmitter) and consequently all functions for working with this module contain the common root “USART” in their name. Actually, according to PIC’s datasheet the discussed module is named EUSART (enhanced USART), however the word “enhanced” has been dropped from eProDas terminology since it implies certain comparison with something inferior (like earlier implementations of USART module) and accordingly does not make much sense in our case.

The communication between PIC and external asynchronous serial periphery is done over two PIC’s pins RC7/RX and RC6/TX. The former is devoted for reception of data from periphery whereas the latter transmits bits from PIC to peripheral units. (If you are new to communication busses: there is no separate clock signal since the point of the word “asynchronous” in the name of transfer is that the clock is implicitly encoded into data signal.)

**Note.** Pin RC7/RX of USART module is shared with SPI module, where the role of the pin is RC7/SDO. Such selection is extremely unfortunate since the pin must be configured as digital input to fulfil its USART role, but it must be digital output for proper working of SPI module. If you use both of these modules in your application, please read section 16.6, which explains how eProDas can help you living with this idiosyncrasy.

As it is more or less the case with all the so far described modules, USART module needs to be properly configured before it can serve you. Utilize the following function to configure USART for RS-232 transfer or something else that fits into the RS-232 scheme.

#### 15.7.1.1 Config_USART_RS232

Prototypes in C/C++

```
int eProDas_Config_USART_RS232(unsigned int DeviceIdx, unsigned int BaudRate,
  unsigned int EnableInput, unsigned int EnableOutput);
```

Prototypes in Delphi

```
function eProDas_Config_USART_RS232(DeviceIdx: LongWord; BaudRate: LongWord;
  EnableInput: LongWord; EnableOutput: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_Config_USART_RS232(ByVal DeviceIdx As UInteger,
  ByVal BaudRate As UInteger, ByVal EnableInput As UInteger,
  ByVal EnableOutput As UInteger) As Integer
```

The configuration is rather straightforward. You only need to specify `BaudRate` of the transfer (like 9600), where the selected baud rate is the same for transmission and reception of bytes. Not all baud rates are possible, however; please read the section 15.7.1.5 for details or check the actual outcome with the function `GetBaudRate` (section 15.7.1.4).

With non-zero parameters `EnableInput` and `EnableOutput` you instructs the device that you intend to receive and transmit the data over USART module, respectively. These settings affect configuration of I/O pins according to your selection. Specifically, if you set the parameter `EnableInput` to zero, the input pin RC7/RX is not configured as digital input, as it is needed for proper USART reception. Similarly, a zero value of the parameter `EnableOutput` prevents configuration of pin RC6/TX as digital output, as it is needed for transmission over USART module.

### 15.7.1.2 TurnOff_USART

Prototypes in C/C++

```
int eProDas_TurnOff_USART(unsigned int DeviceIdx);
```

Prototypes in Delphi

```
function eProDas_TurnOff_USART(DeviceIdx: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_TurnOff_USART(ByVal DeviceIdx As UInteger) As Integer
```

When you no longer need USART services, turn off USART module with this very function.

### 15.7.1.3 USART_SetBaudRate

Prototypes in C/C++

```
int eProDas_USART_SetBaudRate(unsigned int DeviceIdx,
  unsigned int BaudRate);
```

Prototypes in Delphi

```
function eProDas_USART_SetBaudRate(DeviceIdx: LongWord;
  BaudRate: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_USART_SetBaudRate(ByVal DeviceIdx As UInteger,
  ByVal BaudRate As UInteger) As Integer
```

With this function you can change baud rate of USART transfer on the fly. All other settings are preserved.

**Note.** Not all baud rates are possible. Please read the section 15.7.1.5 for details or check the actual outcome with the function `GetBaudRate`.

### 15.7.1.4 USART_GetBaudRate

Prototypes in C/C++

```
int eProDas_USART_GetBaudRate(unsigned int DeviceIdx,
  unsigned int &BaudRate);
```

Prototypes in Delphi

```
function eProDas_USART_GetBaudRate(DeviceIdx: LongWord;
  var BaudRate: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_USART_GetBaudRate(ByVal DeviceIdx As UInteger,
  ByRef BaudRate As UInteger) As Integer
```

This function reports the current baud rate settings. Although such operation may seem superfluous you may still want to perform it. Namely, USART module is not capable of synthesizing all baud rates that you may think of, so by calling this function after using the function `SetBaudRate` or `Config_USART_XXX` you can check the actual resulting state of baud rate configuration.

**Note.** Baud rate is returned as integer number of bits per second, although generally the exact baud rate also contains a fractional part. For example, if returned baud rate is 9 600 there is a possibility that the exact baud rate is, say, 9 600.3. Such imprecision should not be important if your USART device's transfer implementation is robust enough, since asynchronous transfers are designed with such tolerances in mind and generally they can cope with fair baud rate variations without problems.

### 15.7.1.5 USART_SetBaudRateRaw

Prototypes in C/C++

```
int eProDas_USART_SetBaudRateRaw(unsigned int DeviceIdx, unsigned int Divider,
  unsigned int HighSpeed);
```

Prototypes in Delphi

```
function eProDas_USART_SetBaudRateRaw(DeviceIdx: LongWord; Divider: LongWord;
  HighSpeed: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_SetBaudRateRaw(ByVal DeviceIdx As UInteger,
  ByVal Divider As UInteger, ByVal HihghSpeed As UInteger) As Integer
```

This is the actual function that does the baud rate configuration. Under the hood functions `SetBaudRate` as well as all functions `Config_USART_XXX` rely on `SetBaudRateRaw` to do the job.

Internally, PIC synthesizes baud rate for USART transfer by dividing 48 MHz clock with a dedicated counter of settable 16 bit period (parameter `Divider`) and configurable prescaler. When parameter `HighSpeed` holds a non-zero value, the prescaler of 4 is selected otherwise the value of 16 applies. The actual baud rates can be calculated according to the following formulas:

- if `HighSpeed`=0: $$\text{BaudRate} = \frac{48\text{ MHz}}{16\cdot(\texttt{Divider}+1)},$$
- if `HighSpeed`≠0: $$\text{BaudRate} = \frac{48\text{ MHz}}{4\cdot(\texttt{Divider}+1)}.$$

**Example.** According to the last formula, by selecting `HighSpeed` mode of operation and `Divider` of 1249, baud rate of 9 600 is achieved. Of course, normally you rely on functions `SetBaudRate` or `Config_USART_XXX` to calculate these parameters for you.

**Note.** Since period of PIC's baud rate synthesizer is 16-bit wide, the parameter `Divider` must be between 0 and 65535. If such divider in conjunction with prescaler of 4 or 16 cannot synthesize baud rate of your choice you are forced to resort to the set of values that are not beyond the reality according to the above formulas.

### 15.7.2 Transferring data over USART bus

This section presents the basic subset of eProDas functions for transferring single- and multi-byte data packets over USART bus. Although this seems to be everything you will ever need, we would nonetheless like to refer you to the already mentioned `Transfer` backbone (section 23.2), which enables you to implement various USART transferring masterpieces.

#### 15.7.2.1 USART_TransferByte group of functions

Prototypes in C/C++

```
int eProDas_USART_TransferByte(unsigned int DeviceIdx, unsigned char WriteByte,
  unsigned char &ReadByte);

int eProDas_USART_TransferByteOUT(unsigned int DeviceIdx, unsigned char WriteByte);

int eProDas_USART_TransferByteIN(unsigned int DeviceIdx, unsigned char &ReadByte);
```

Prototypes in Delphi

```
function eProDas_USART_TransferByte(DeviceIdx: LongWord; WriteByte: Byte;
  var ReadByte: Byte):integer;

function eProDas_USART_TransferByteOUT(DeviceIdx: LongWord;
  WriteByte: Byte):integer;

function eProDas_USART_TransferByteIN(DeviceIdx: LongWord;
  var ReadByte: Byte):integer;
```

Prototypes in VisualBasic

```
Function eProDas_USART_TransferByte(ByVal DeviceIdx As UInteger,
  ByVal WriteByte As Byte, ByRef ReadByte As Byte) As Integer

Function eProDas_USART_TransferByteOUT(ByVal DeviceIdx As UInteger,
  ByVal WriteByte As Byte) As Integer

Function eProDas_USART_TransferByteIN(ByVal DeviceIdx As UInteger,
  ByRef ReadByte As Byte) As Integer
```

With plain `TransferByte` you perform full duplex one byte USART transfer, whereas functions with suffixes `OUT` or `IN` perform only transmission or reception, respectively.

**Note.** The full duplex and `IN` functions do not return until the byte is received; if reception does not happen your eProDas device waits for it forever.

#### 15.7.2.2 USART_TransferArray group of functions

Prototypes in C/C++

```
int eProDas_USART_TransferArray(unsigned int DeviceIdx, unsigned char *WriteArray,
  unsigned char *ReadArray, unsigned int Length);

int eProDas_USART_TransferArrayOUT(unsigned int DeviceIdx,
  unsigned char *WriteArray, unsigned int Length);

int eProDas_USART_TransferArrayIN(unsigned int DeviceIdx, unsigned char *ReadArray,
  unsigned int Length);
```

Prototypes in Delphi

```
function eProDas_USART_TransferArray(DeviceIdx: LongWord; WriteArray: PByte;
  ReadArray: PByte; Length: LongWord):integer;

function eProDas_USART_TransferArrayOUT(DeviceIdx: LongWord; WriteArray: PByte;
  Length: LongWord):integer;

function eProDas_USART_TransferArrayIN(DeviceIdx: LongWord; ReadArray: PByte;
  Length: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_USART_TransferArray(ByVal DeviceIdx As UInteger,
  ByRef WriteArray As Byte, ByRef ReadArray As Byte, ByVal Length As UInteger)
  As Integer

Function eProDas_USART_TransferArrayOUT(ByVal DeviceIdx As UInteger,
  ByRef WriteArray As Byte, ByVal Length As UInteger) As Integer

Function eProDas_USART_TransferArrayIN(ByVal DeviceIdx As UInteger,
  ByRef ReadArray As Byte, ByVal Length As UInteger) As Integer
```

With plain `TransferArray` you perform full duplex multi byte USART transfer, whereas functions with suffixes `OUT` or `IN` perform transmission or reception, respectively. In the case of plain `TransferArray` the maximal possible array length is 16 bytes, whereas the other two functions can cope with arrays of up to 31 bytes.

In the case of full duplex transfer the transmission and reception length is the same. If this is prohibitively limiting, you can always combine two function calls to achieve independent transfer lengths. For example, if you want to transmit 15 bytes of data and receive 9 of them, use plain `TransferArray` function for 9 bytes transfer in conjunction with additional call to function `TransferArrayOUT` for transmission of remaining 6 bytes.

**Note 1.** Calling functions `TransferArrayOUT` and `TransferArrayIN` in sequence is not the same as relying on the function `TransferArray` for the task, since during transmission, received data is not stored in a buffer and therefore only one byte of data, which is buffered by PIC's hardware, can be received during the whole multi-byte OUT transfer process.

**Note 2.** The full duplex and `IN` functions do not return until all demanded bytes are received; if sufficient data does not arrive your eProDas device waits for it forever.

#### 15.7.2.3 USART_ClearReceptionBuffer

Prototypes in C/C++

```
int eProDas_USART_ClearReceptionBuffer(unsigned int DeviceIdx);
```

Prototypes in Delphi

```
function eProDas_USART_ClearReceptionBuffer(DeviceIdx: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_USART_ClearReceptionBuffer(ByVal DeviceIdx As UInteger) As Integer
```

When working with USART transfers you may appreciate this function that clears PIC's hardware reception buffer of USART module. As it was already mentioned, PIC possesses one byte buffer, to which it captures a byteful of data that it receives over USART bus. It may be a good idea to clear this buffer before initiating USART transfer protocol, since the buffer may contain some spurious data that it received somewhere in the past before the meaningful data started to arrive. If such data is not deleted prior to initiating the transfer, your eProDas application would think that it is a part of received data. Further, since you always specify the amount of received data, the last useful byte would remain unread due to the reception of one false byte.

# 16 Software implemented features

This chapter describes functions that implement certain eProDas features purely in software and therefore extend the previously described functions for accessing the PIC's or externally added hardware. The aim of implementation of these features is to simplify development of eProDas application by providing as rich spectrum of ready-to-exploit functionality as possible.

## 16.1 Operations on digital I/O ports

These functions extend the functionality of functions in section 15.1.

### 16.1.1 AND_Port, OR_Port, XOR_Port, XORXOR_Port

Prototypes in C/C++

```
int eProDas_AND_Port(unsigned int DeviceIdx, unsigned int Port,
  unsigned char AND_Value);

int eProDas_OR_Port(unsigned int DeviceIdx, unsigned int Port,
  unsigned char OR_Value);

int eProDas_XOR_Port(unsigned int DeviceIdx, unsigned int Port,
  unsigned char XOR_Value);

int eProDas_XORXOR_Port(unsigned int DeviceIdx, unsigned int Port,
  unsigned char XORXOR_Value);
```

Prototypes in Delphi

```
function eProDas_AND_Port(DeviceIdx: LongWord; Port: LongWord;
  AND_Value: Byte):integer;

function eProDas_OR_Port(DeviceIdx: LongWord; Port: LongWord;
  OR_Value: Byte):integer;

function eProDas_XOR_Port(DeviceIdx: LongWord; Port: LongWord;
  XOR_Value: Byte):integer;

function eProDas_XORXOR_Port(DeviceIdx: LongWord; Port: LongWord;
  XORXOR_Value: Byte):integer;
```

Prototypes in VisualBasic

```
Function eProDas_AND_Port(ByVal DeviceIdx As UInteger, ByVal Port As UInteger,
  ByVal AND_Value As Byte) As Integer

Function eProDas_OR_Port(ByVal DeviceIdx As UInteger, ByVal Port As UInteger,
  ByVal OR_Value As Byte) As Integer

Function eProDas_XOR_Port(ByVal DeviceIdx As UInteger, ByVal Port As UInteger,
  ByVal XOR_Value As Byte) As Integer

Function eProDas_XORXOR_Port(ByVal DeviceIdx As UInteger, ByVal Port As UInteger,
  ByVal XORXOR_Value As Byte) As Integer
```

This group of functions work in the same way as the function `WritePort` (section 15.1.15) does, except that these functions do not write the user specified value directly to the port's latch register but they instead perform certain operation between the current latch value and the user specified one.

Functions `AND`, `OR` and `XOR` should be self evident; they perform bitwise AND, OR and XOR between the current latch value and the value of the third function parameter. Function `XORXOR` does the XOR operation twice which results in the same latch value as it was at the beginning, however the transition of states where XOR mask is 1 is evident on the pins by means of which it is possible to generate short pulses (approximately 83 ns), for example to trigger clock signal or to make external latch transparent for a short period of time.

### 16.1.2 AND_Ports, OR_Ports, XOR_Ports, XORXOR_Ports

Prototypes in C/C++

```
int eProDas_AND_Ports(unsigned int DeviceIdx, unsigned char *AND_Values);

int eProDas_OR_Ports(unsigned int DeviceIdx, unsigned char *OR_Values);

int eProDas_XOR_Ports(unsigned int DeviceIdx, unsigned char *XOR_Values);

int eProDas_XORXOR_Ports(unsigned int DeviceIdx, unsigned char *XORXOR_Values);
```

Prototypes in Delphi

```
function eProDas_AND_Ports(DeviceIdx: LongWord; AND_Values: PByte):integer;

function eProDas_OR_Ports(DeviceIdx: LongWord; OR_Values: PByte):integer;

function eProDas_XOR_Ports(DeviceIdx: LongWord; XOR_Values: PByte):integer;

function eProDas_XORXOR_Ports(DeviceIdx: LongWord; XORXOR_Values: PByte):integer;
```

Prototypes in VisualBasic

```
Function eProDas_AND_Ports(ByVal DeviceIdx As UInteger, ByRef AND_Values As Byte)
  As Integer

Function eProDas_OR_Ports(ByVal DeviceIdx As UInteger, ByRef OR_Values As Byte)
  As Integer

Function eProDas_XOR_Ports(ByVal DeviceIdx As UInteger, ByRef XOR_Values As Byte)
  As Integer

Function eProDas_XORXOR_Ports(ByVal DeviceIdx As UInteger,
  ByRef XORXOR_Values As Byte) As Integer
```

These functions parallel previous group of functions and they perform certain bitwise operation on all port's latch registers and user specified values. Contrary to discouraged utilization of the function `WritePorts` (section 15.1.16) we can again recommend to use this group of functions when working with two or more ports instead of performing several isolated calls to the appropriate function in the section 16.1.1, since AND, OR or XOR bitmask can always be arranged in such a way that they do not affect the values (i.e. bits/pins) that you do not want to modify.

### 16.1.3 ROTx_Port, SHIFTx_Port, Complement, Negate, SwapNibbles

Prototypes in C/C++

```
int eProDas_ROTL_Port(unsigned int DeviceIdx, unsigned int Port);

int eProDas_ROTR_Port(unsigned int DeviceIdx, unsigned int Port);

int eProDas_SHIFTL_Port(unsigned int DeviceIdx, unsigned int Port,
  unsigned int LSB_Value);

int eProDas_SHIFTR_Port(unsigned int DeviceIdx, unsigned int Port,
  unsigned int MSB_Value);

int eProDas_Complement_Port(unsigned int DeviceIdx, unsigned int Port);

int eProDas_Negate_Port(unsigned int DeviceIdx, unsigned int Port);

int eProDas_SwapNibbles_Port(unsigned int DeviceIdx, unsigned int Port);
```

Prototypes in Delphi

```
function eProDas_ROTL_Port(DeviceIdx: LongWord; Port: LongWord):integer;

function eProDas_ROTR_Port(DeviceIdx: LongWord; Port: LongWord):integer;

function eProDas_SHIFTL_Port(DeviceIdx: LongWord; Port: LongWord;
  LSB_Value: LongWord):integer;

function eProDas_SHIFTR_Port(DeviceIdx: LongWord; Port: LongWord;
  MSB_Value: LongWord):integer;

function eProDas_Complement_Port(DeviceIdx: LongWord; Port: LongWord):integer;

function eProDas_Negate_Port(DeviceIdx: LongWord; Port: LongWord):integer;

function eProDas_SwapNibbles_Port(DeviceIdx: LongWord; Port: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_ROTL_Port(ByVal DeviceIdx As UInteger, ByVal Port As UInteger)
  As Integer

Function eProDas_ROTR_Port(ByVal DeviceIdx As UInteger, ByVal Port As UInteger)
  As Integer

Function eProDas_SHIFTL_Port(ByVal DeviceIdx As UInteger, ByVal Port As UInteger,
  ByVal LSB_Value As Integer) As Integer

Function eProDas_SHIFTR_Port(ByVal DeviceIdx As UInteger, ByVal Port As UInteger,
  ByVal MSB_Value As Integer) As Integer

Function eProDas_Complement(ByVal DeviceIdx As UInteger, ByVal Port As UInteger)
  As Integer

Function eProDas_Negate_Port(ByVal DeviceIdx As UInteger, ByVal Port As UInteger)
  As Integer

Function eProDas_SwapNibbles(ByVal DeviceIdx As UInteger, ByVal Port As UInteger)
  As Integer
```

Functions `ROTL`/`ROTR` rotate port's latch value one bit left/right, whereas `SHIFTL`/`SHIFTR` perform 1 bit shift left/right. When shifting port an additional parameter specifies the new value of the LSB/MSB bit that would otherwise be undefined.

Functions `Complement/Negate` perform one's/two's complement of the port's latch value, whereas `SwapNibbles` exchange bits 0..3 with bits 4..7 and vice versa.

### 16.1.4 ADD_Port, SUB_Port, INC_Port, DEC_Port

Prototypes in C/C++

```
int eProDas_ADD_Port(unsigned int DeviceIdx, unsigned int Port,
  unsigned char ADD_Value);

int eProDas_SUB_Port(unsigned int DeviceIdx, unsigned int Port,
  unsigned char SUB_Value);

int eProDas_INC_Port(unsigned int DeviceIdx, unsigned int Port);

int eProDas_DEC_Port(unsigned int DeviceIdx, unsigned int Port);
```

Prototypes in Delphi

```
function eProDas_ADD_Port(DeviceIdx: LongWord; Port: LongWord;
  ADD_Value: Byte):integer;

function eProDas_SUB_Port(DeviceIdx: LongWord; Port: LongWord;
  SUB_Value: Byte):integer;

function eProDas_INC_Port(DeviceIdx: LongWord; Port: LongWord):integer;

function eProDas_DEC_Port(DeviceIdx: LongWord; Port: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_ADD_Port(ByVal DeviceIdx As UInteger, ByVal Port As UInteger,
  ByVal ADD_Value As Byte) As Integer

Function eProDas_SUB_Port(ByVal DeviceIdx As UInteger, ByVal Port As UInteger,
  ByVal SUB_Value As Byte) As Integer

Function eProDas_INC_Port(ByVal DeviceIdx As UInteger, ByVal Port As UInteger)
  As Integer

Function eProDas_DEC_Port(ByVal DeviceIdx As UInteger, ByVal Port As UInteger)
  As Integer
```

Here is a more arithmetic oriented group of functions. Function `ADD_Port` adds a user specified value to the current latch register, whereas `SUB_Port` subtracts the value by adding two's complement to the current latch register. As shorthand for adding or subtracting 1 from the value there exist functions `INC_Port` and `DEC_Port` for incrementing and decrementing the port value, respectively.

### 16.1.5 GeneratePulses

Prototypes in C/C++

```
int eProDas_GeneratePulses(unsigned int DeviceIdx, unsigned int Port,
  unsigned int Pin, unsigned int NumberOfDelays, unsigned int *Delays);
```

Prototypes in Delphi

```
function eProDas_GeneratePulses(DeviceIdx: LongWord; Port: LongWord; Pin: LongWord;
  NumberOfDelays: LongWord; Delays: PLongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_GeneratePulses(ByVal DeviceIdx As UInteger,
  ByVal Port As UInteger, ByVal Pin As UInteger, ByVal NumberOfDelays As UInteger,
  ByRef Delays As UInteger) As Integer
```

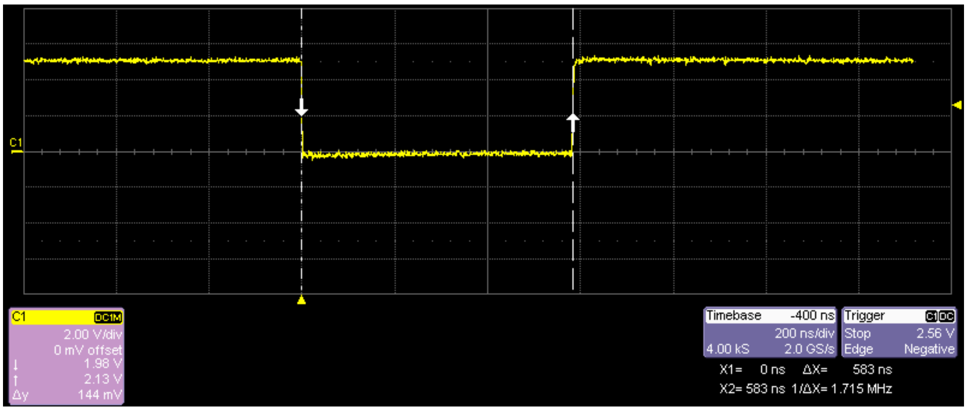
From time to time a need arises for generating a pulse or pulse train of prescribed duration(s). Fortunately, eProDas enables you to do the task easily. Initially, the function `GeneratePulses` toggles the state of the specified pin. After that it toggles the same pin again for the `NumberOfDelays` number of times (max. 40) where the delay of each toggle is specified by the array `Delays`. Each entry of the array must be between 1 and 256. The actual delay is specified by the following equation.

$$\text{delay}\left[\text{ns}\right] = (4 + 3 \cdot \mathit{Delay}) \cdot \frac{1000}{12}$$

The *Delay* is a user specified value of the delay in the array `Delays`. The delay of the next pin toggle is a sum of execution times of PIC instructions (at 12 MHz execution rate) that constitute delay loop. It takes 3 instruction's time spans to execute one loop which results in multiplication of *Delay* by 3; plus there are four instruction's time spans to toggle the pin and prepare for the processing of a new delay.

**Note.** There exists a file `pulses.PDF` in eProDas documentation directory with calculated delays for convenient lookup and determination of a proper *Delay* value for a required delay generation.

**Example.** The shortest possible pulse is specified by the *Delay* of 1 which results in nominal pulse length of 583.33 ns according to the above equation. If we assume that the port state was in logic 1 (for example by using the function `SetPin`) prior to calling the function `GeneratePulses`, the figure on your oscilloscope would look something like this.



C1 DC1M
2.00 V/div
0 mV offset
↓ 1.98 V
↑ 2.13 V
Δy 144 mV

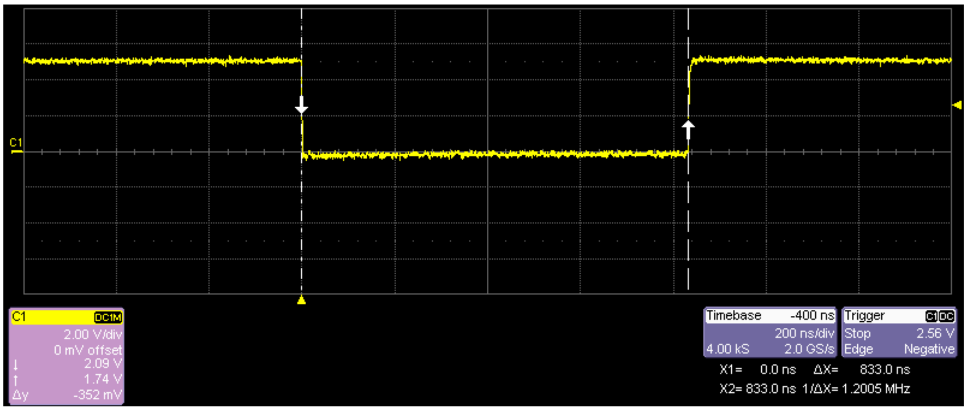
Timebase -400 ns
200 ns/div
4.00 kS 2.0 GS/s

Trigger C1 DC
Stop 2.56 V
Edge Negative

X1= 0 ns ΔX= 583 ns
X2= 583 ns 1/ΔX= 1.715 MHz

**Figure 42: Observing the pulse on the oscilloscope screen.**

If we prolong the *Delay* to 2, the length of the pulse increases to 833.33 ns as it is evident from the following figure.



C1 DC1M
2.00 V/div
0 mV offset
↓ 2.09 V
↑ 1.74 V
Δy -352 mV

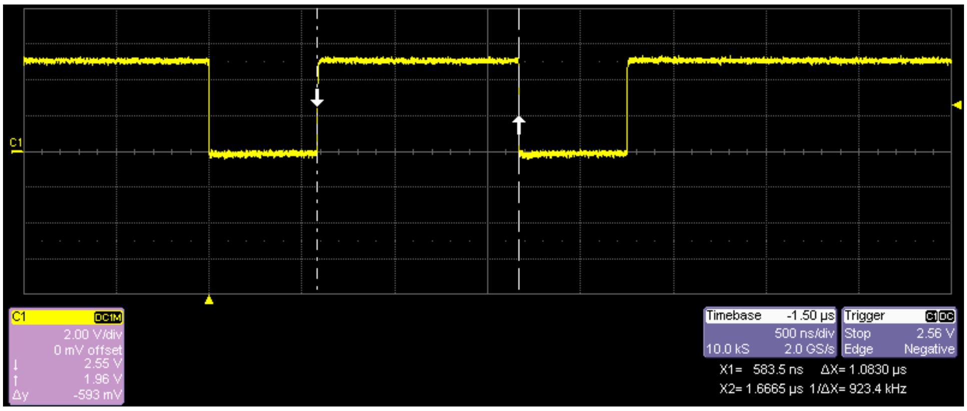
Timebase -400 ns
200 ns/div
4.00 kS 2.0 GS/s

Trigger C1 DC
Stop 2.56 V
Edge Negative

X1= 0.0 ns ΔX= 833.0 ns
X2= 833.0 ns 1/ΔX= 1.2005 MHz

**Figure 43: Observing the prolonged pulse on the oscilloscope screen.**

For the final example, let’s specify three toggling of pin, which are determined by the *Delay*s of 1, 3 and 1, respectively. The result is presented in the following figure.



C1 DC1M
2.00 V/div
0 mV offset
↓ 2.55 V
↑ 1.96 V
Δy -593 mV

Timebase -1.50 µs
500 ns/div
10.0 kS 2.0 GS/s

Trigger C1DC
Stop 2.56 V
Edge Negative

X1= 583.5 ns ΔX= 1.0830 µs
X2= 1.6665 µs 1/ΔX= 923.4 kHz

**Figure 44: Observing the sequence of one plus three pin toggles.**

As you can see, after the initial toggle the pin returns to the state logic 1 after 583 ns passes. Then it is toggled back to the state logic 0 for the second time 1083 ns after the previous transition, and it is finally returned to the logic 1 after additional 583 ns.

**Note.** If you specify an even number of delays, the end state of the pin will be different from the initial one, since an odd number of toggles would occur (the initial one plus one for each specified delay).

### 16.1.6 GenerateLongPulses

Prototypes in C/C++

```
int eProDas_GenerateLongPulses(unsigned int DeviceIdx, unsigned int Port,
  unsigned int Pin, unsigned int NumberOfDelays, unsigned int *Delays);
```

Prototypes in Delphi

```
function eProDas_GenerateLongPulses(DeviceIdx: LongWord; Port: LongWord;
  Pin: LongWord; NumberOfDelays: LongWord; Delays: PLongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_GenerateLongPulses(ByVal DeviceIdx As UInteger,
  ByVal Port As UInteger, ByVal Pin As UInteger, ByVal NumberOfDelays As UInteger,
  ByRef Delays As UInteger) As Integer
```

Occasionally, you will need to generate longer pulses than the previous function allows you to (about 64 µs maximum) and on such occasions the function `GenerateLongPulses` comes to the rescue. The entries of the array `Delays` are now allowed to fall in the interval from 1 to 65 536; however the maximal number of delays dropped from 40 to 20.

The actual delay is somewhat more complicated to calculate than before. Let’s have fingers crossed that the next formula will not scary you.

$$\text{delay}\left[\text{ns}\right] = \left(10 + 3 \cdot Delay + 2 \cdot \left\lfloor \frac{(Delay - 1)}{256} \right\rfloor\right) \cdot \frac{1000}{12}$$

The increased complication is due to the fact that there are now two program loops (one inside another). Dealing with two loops is somewhat more time consuming so there are now 10 instructions of a fixed amount of delay in comparison with 4, as it was the case before. The inner loop which counts the least significant byte of the specified delay takes again 3 instruction cycles to execute. But now there is also the outer loop which executes as many times as the most significant byte of the delay determines; this contribution is taken into account with the last term in the brackets.

**Note.** There exists a file `long_pulses.PDF` in eProDas documentation directory with calculated delays for a lookup and determination of a proper *Delay* value for a required delay generation.

**Example.** When specifying the *Delay* of 1, 2 and 3 the resulting delays are 1 083 ns, 1 333 ns and 1 583 ns, respectively. As you can see, each increase of *Delay* prolongs the delay for 3 instructions at 12 MHz or 250 ns. This pattern lasts up to the *Delay* of 256, which results in 64 833 ns of delay.

The pattern is broken with the *Delay* of 257 which forces an outer loop to execute one additional time and contributes two additional instructions to the scheme. The actual delay is now 65 250 ns and not 65 083 as it would be if the increase of delay was uniform. *Delay*s of 258, 259 and 512 result in 65 550 ns, 65 750 ns and 129 µs, respectively. Then at 513 the outer loop executes for one more time and the resulting delay is 129.417 µs. The maximal *Delay* of 65 536 results in the longest possible delay of 16.427 ms.

### 16.1.7 WaitPortState, WaitPortTransition

Prototypes in C/C++

```
eProDasDLL_API int eProDas_WaitPortState(unsigned int DeviceIdx, unsigned int Port,
  unsigned int AND_Mask, unsigned int State);
```

```
eProDasDLL_API int eProDas_WaitPortTransition(unsigned int DeviceIdx,
  unsigned int Port, unsigned int AND_Mask, unsigned int State1,
  unsigned int State2);
```

Prototypes in Delphi

```
function eProDas_WaitPortState(DeviceIdx: LongWord; Port: LongWord;
  AND_Mask: LongWord; State: LongWord):integer;
```

```
function eProDas_WaitPortTransition(DeviceIdx: LongWord; Port: LongWord;
  AND_Mask: LongWord; State1: LongWord; State2: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_WaitPortState(ByVal DeviceIdx As UInteger, ByVal Port As UInteger,
  ByVal AND_Mask As UInteger, ByVal State As UInteger) As Integer
```

```
Function eProDas_WaitPortTransition(ByVal DeviceIdx As UInteger,
  ByVal Port As UInteger, ByVal AND_Mask As UInteger, ByVal State1 As UInteger,
  ByVal State2 As UInteger) As Integer
```

Often you need to wait for a certain state on a port. For example, your external AD converter or Bluetooth module indicates that AD result or new received data is ready for your application to chew. In such cases when your application has nothing else to do it can wait for the event that it longs for by using one of the two functions that are just being described.

Function `WaitPortState` waits for certain bit pattern to appear on the chosen port. Since generally not all pins are relevant to certain wait condition there is a parameter `AND_Mask` to discard irrelevant pins, which you do by setting the respective bits of the mask to zero. The function does nothing else but runs in a tight loop reading port value, applying bitwise AND mask and waiting that the result equals the `State` value, which is again subject to AND mask.

**Note 1.** When using this function make sure that the state you are waiting for is going to come along or else your eProDas device will wait for it till the end of its power supply time.

**Note 2.** Observation of pin state is done purely in software. This means that the sought after state should be presented on the port for a long enough time that it is certainly observed by PIC. The loop that monitors the state takes 5 instruction cycles at 83 ns to execute, which means that in worst case the state is going to be observed only after 417 ns passes. However, due to the asynchronous nature of the process there can be very little stated about how short a pulse must be in order to not to be observed for sure. If your design leads to short spikes on pins you may be in trouble if things are arranged in a bad way.

Instead of waiting for a certain state you can also wait for a transition from one predetermined state to the other one, which is done by the function `WaitPortTransition`. Somewhat oversimplified but clarifying description would be that this function works semantically as two separate calls to the function `WaitPortState` where for the first call the `State` value equals `State1` and for the second call it is set to the `State2`.

**Note 3.** Due to the above description the function will successfully recognize the transition even if there are other states on the port between the appearance of states `State1` and `State2`. In the first step the function simply waits for `State1` and when it encounters it there enters the second wait phase where a wait on `State2` is being done.

**Note 4.** For either state to be recognized all pins have to match their `State1` or `State2` value. If isolated pins transit from their states in an uncorrelated way the function may never succeed in observing the proper state although individual pins made a prescribed transition.

**Note 5.** Above notes 1 and 2 still apply.

### 16.1.8 RepetitivelyWaitPortState, RepetitivelyWaitPortTransition

Prototypes in C/C++

```
eProDasDLL_API int eProDas_RepetitivelyWaitPortState(unsigned int DeviceIdx,
  unsigned int Repetitions, unsigned int Port, unsigned int AND_Mask,
  unsigned int State);

eProDasDLL_API int eProDas_RepetitivelyWaitPortTransition(unsigned int DeviceIdx,
  unsigned int Repetitions, unsigned int Port, unsigned int AND_Mask,
  unsigned int State1, unsigned int State2);
```

Prototypes in Delphi

```
function eProDas_RepetitivelyWaitPortState(DeviceIdx: LongWord;
  Repetitions: LongWord; Port: LongWord; AND_Mask: LongWord;
  State: LongWord):integer;

function eProDas_RepetitivelyWaitPortTransition(DeviceIdx: LongWord;
  Repetitions: LongWord; Port: LongWord; AND_Mask: LongWord; State1: LongWord;
  State2: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_RepetitivelyWaitPortState(ByVal DeviceIdx As UInteger,
  ByVal Repetitions As UInteger, ByVal Port As UInteger,
  ByVal AND_Mask As UInteger, ByVal State As UInteger) As Integer

Function eProDas_RepetitivelyWaitPortTransition(ByVal DeviceIdx As UInteger,
  ByVal Repetitions As UInteger, ByVal Port As UInteger,
  ByVal AND_Mask As UInteger, ByVal State1 As UInteger, ByVal State2 As UInteger)
  As Integer
```

This is an extension of functions `WaitPortState` and `WaitPortTransition` whereby the prescribed state(s) is(are) acknowledged only after it is(they are) read several times (the `Repetitions` number of times, from 1 to 65 535) in a row with the same outcome. For example, if `Repetitions` equals 50,000 the demanded state would have to be sequentially read for 50,000 times before the function returns; if after 49,999 readings or anywhere else the state changes, the counting starts all over again.

There is a delay of about 6 PIC instructions (500 ns) between two adjacent readings of the state, so by specifying the maximal possible number of repetitions it is possible to demand that the state is unchanged (more precisely: that the changes are not detected) for more than 30 ms before it is declared stable. This feature is meant as a poor man's filter of digital inputs. If the number of repetitions is fairly high, then noise and spikes on the line have a much less chance to be recognized as the desired state. The feature is especially suitable for debouncing of mechanical keys, e.g. if your application counts the number that a certain key was pressed, you will surely love this function.

**Note.** All notes from section 16.1.7 still apply.

### 16.1.9 WaitStablePortState

Prototypes in C/C++

```
int eProDas_WaitStablePortState(unsigned int DeviceIdx, unsigned int Repetitions,
  unsigned int Port, unsigned int AND_Mask, unsigned int &ReachedState);
```

Prototypes in Delphi

```
function eProDas_WaitStablePortState(DeviceIdx: LongWord; Repetitions: LongWord;
  Port: LongWord; AND_Mask: LongWord; var ReachedState: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_WaitStablePortState(ByVal DeviceIdx As UInteger,
  ByVal Repetitions As UInteger, ByVal Port As UInteger,
  ByVal AND_Mask As   UInteger, ByRef ReachedState As UInteger) As Integer
```

Instead of waiting for a prescribed state you may want to wait for any stabilized state. This function works in the same way as the function `RepetitivelyWaitPortState` does, except that you do not specify a state to wait for. As soon as the port reading returns `Repetitions` equal values in a sequence, the function returns and the state is reported to you through the parameter `ReachedState`.

### 16.1.10 WaitPinState, WaitPinTransition

Prototypes in C/C++

```
eProDasDLL_API int eProDas_WaitPinState(unsigned int DeviceIdx, unsigned int Port,
  unsigned int Pin, unsigned int State);

eProDasDLL_API int eProDas_WaitPinTransition(unsigned int DeviceIdx,
  unsigned int Port, unsigned int Pin, unsigned int LowToHigh);
```

Prototypes in Delphi

```
function eProDas_WaitPinState(DeviceIdx: LongWord; Port: LongWord; Pin: LongWord;
  State: LongWord):integer;

function eProDas_WaitPinTransition(DeviceIdx: LongWord; Port: LongWord;
  Pin: LongWord; LowToHigh: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_WaitPinState(ByVal DeviceIdx As UInteger, ByVal Port As UInteger,
  ByVal Pin As UInteger, ByVal State As UInteger) As Integer

Function eProDas_WaitPinTransition(ByVal DeviceIdx As UInteger,
  ByVal Port As UInteger, ByVal Pin As UInteger, ByVal LowToHigh As UInteger)
  As Integer
```

Instead of waiting for an entire port state you will more often want to wait for a prescribed state of a selected pin which is done by the two functions `WaitPinState` and `WaitPinTransition`. Under the hood these two functions rely on the two functions from the section 16.1.7 (please, read the useful notes).

When calling the function `WaitPinState` any non-zero value of the `State` parameter leads to waiting for logic 1 on the pin.

When calling the function `WaitPinTransition` a non-zero value of the `LowToHigh` parameter leads to first waiting for logic 0 state on the pin and then waiting for logic 1 on the same pin. In the opposite case logic 1 must occur first and logic 0 after it.

### 16.1.11 RepetitivelyWaitPinState, RepetitivelyWaitPinTransition

Prototypes in C/C++

```
eProDasDLL_API int eProDas_RepetitivelyWaitPinState(unsigned int DeviceIdx,
  unsigned int Repetitions, unsigned int Port, unsigned int Pin,
  unsigned int State);

eProDasDLL_API int eProDas_RepetitivelyWaitPinTransition(unsigned int DeviceIdx,
  unsigned int Repetitions, unsigned int Port, unsigned int Pin,
  unsigned int LowToHigh);
```

Prototypes in Delphi

```
function eProDas_RepetitivelyWaitPinState(DeviceIdx: LongWord;
  Repetitions: LongWord; Port: LongWord; Pin: LongWord; State: LongWord):integer;

function eProDas_RepetitivelyWaitPinTransition(DeviceIdx: LongWord;
  Repetitions: LongWord; Port: LongWord; Pin: LongWord;
  LowToHigh: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_RepetitivelyWaitPinState(ByVal DeviceIdx As UInteger,
  ByVal Repetitions As UInteger, ByVal Port As UInteger, ByVal Pin As UInteger,
  ByVal State As UInteger) As Integer

Function eProDas_RepetitivelyWaitPinTransition(ByVal DeviceIdx As UInteger,
  ByVal Repetitions As UInteger, ByVal Port As UInteger, ByVal Pin As UInteger,
  ByVal LowToHigh As UInteger) As Integer
```

These functions are a variation of functions in the section 16.1.8, except that they work on an isolated pin instead on the whole port.

### 16.1.12 WaitStablePinState

Prototypes in C/C++

```
int eProDas_WaitStablePinState(unsigned int DeviceIdx, unsigned int Repetitions,
  unsigned int Port, unsigned int Pin, unsigned int &ReachedState);
```

Prototypes in Delphi

```
function eProDas_WaitStablePinState(DeviceIdx: LongWord; Repetitions: LongWord;
  Port: LongWord; Pin: LongWord; var ReachedState: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_WaitStablePinState(ByVal DeviceIdx As UInteger,
  ByVal Repetitions As UInteger, ByVal Port As UInteger, ByVal Pin As UInteger,
  ByRef ReachedState As UInteger) As Integer
```

... an analogous function to the `WaitStablePortState`, except that is monitors an isolated pin instead of the whole port. A non-zero `ReachedState` indicates that the pin was in high logic state at the moment of stable pin state acknowledgment.

### 16.1.13 ConfigureInternalPullupResistors

Prototypes in C/C++

```
int eProDas_ConfigureInternalPullupResistors(unsigned int DeviceIdx,
  unsigned int Enable);
```

Prototypes in Delphi

```
function eProDas_ConfigureInternalPullupResistors(DeviceIdx: LongWord;
  Enable: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_ConfigureInternalPullupResistors(ByVal DeviceIdx As UInteger,
  ByVal Enable As UInteger) As Integer
```

When you want to make your device interactive by providing some keys that the user can press on to walk along menus, confirm certain actions, stop the device in critical situations or whatever, you need to assure a forced and predictable state of the PIC's pin whether the key is pressed or not. In the majority of cases this is done by connecting the pin through a pull-up resistor the opposite supply rail than the key connects it to when it is pressed. As it turns out, PIC already possesses such pull-up resistors internally but only on port B. These resistors can be enabled or disabled by the currently discussed function. Note that you can only enable and disable all pull-up resistors at once. On the positive side, if certain pin of port B is configured as digital output, its pull-up resistor is disabled automatically by the PIC.

## 16.2 Support for implementation of address buses

When working with microprocessors address bus means digital output port that specifies the address of the next to be accessed memory or I/O location in the form of a binary number. This number is then usually decoded by some external circuitry to generate chip-select, chip-enable or similar signals to control communication with connected periphery. The point of address bus is to enable microprocessor to access addressable locations in a random-access way.

Consider a conceptual example in the Figure 45 where pins RB0 to RB2 are devoted for the task of implementing the address bus. The set of pins from RE0 to RE2 seems a proper choice too, but let's say that we also need eight internal AD inputs, which renders these pins reserved. Also, port C with its pins from RC0 to RC2 is extremely suitable for the task, but again let's come up with an excuse that we need output of PWM1 during the process. This time we really need port B to carry the role of address bus for pedagogical reasons.

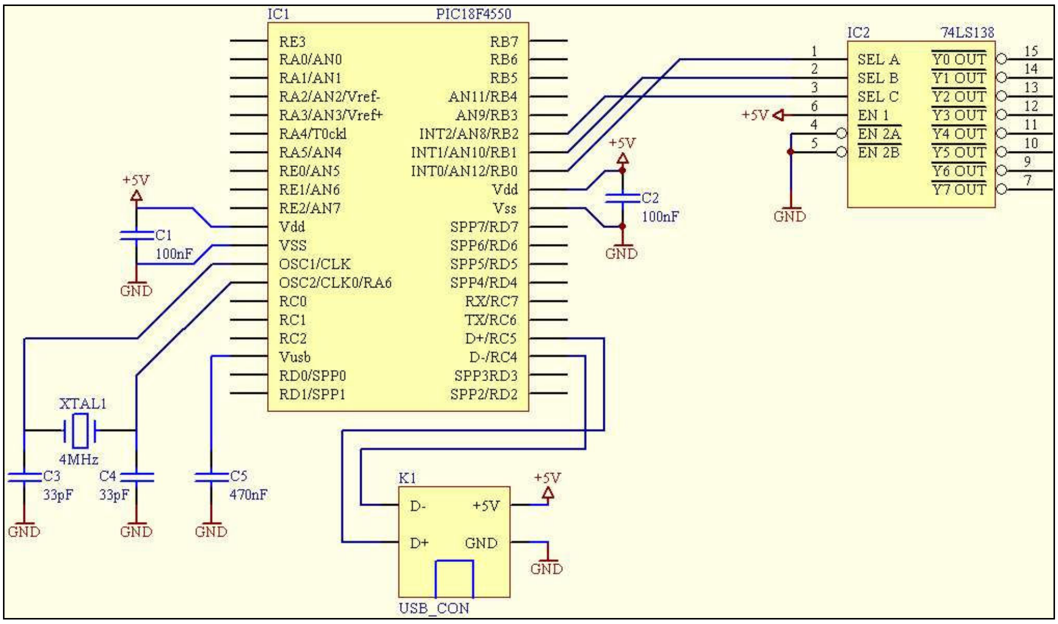


**Figure 45: Conceptual example of address bus implementation.**

With three pins we can address $2^3$=8 peripherals for which we need a suitable decoder like 74HC138, which decodes 3-bit binary number into the active logic state of the associated output pin. To each of the pins from `Y0_OUT` to `Y7_OUT` we can connect e.g. chip-select of one of the external chips or modules with microprocessor interface. Then by outputting, say, binary 4 on pins RB0 to RB2 we would activate the periphery that is hooked to pin `Y4_OUT`. Actually, in many cases only 7 peripherals can be addressed by 3 pins (or generally $2^n$-1 with $n$ pins), since you need one state that does not enable any periphery, for example address 0 could be a “no periphery” by wiring nothing to `Y0_OUT`.
When working with eProDas systems we often do not need random-access liberty, since in the majority of situations your application will probably consist of some number of external circuits (DA, AD, registers, communication circuits, multiplexers...) that you will want to address in a predetermined sequence, often by incrementing address bus value. To implement such a bus we only need a possibility to increment port value, which the already described function `INC_Port` provides.

Well, that is indeed true but we can do better than plain increment can. The problem with using the function `INC_Port` is that it operates on the whole port. How about if we only need to address eight (oops, seven) peripheral modules as the Figure 45 presumes, for which we need only three pins? Will we waste entire port B for the task by using plain increment?

Worse still, there are situations where we need more than one address bus. For example, we may use “the main” 3-bit address bus to access external periphery such as AD and DA converters (by decoding their address by e.g. integrated circuit 74HC138), the “alternative” 2-bit bus to select periphery that is hooked to SPI (serial peripheral interface) bus (using ½ of 74HC139) that needs to operate in parallel with the main bus, and the “channel select” 3-bit bus for determining which of the eight temperature sensors is currently being connected to AD converter (using 74HC4051 or so). By using simple increment we would spend three precious ports only to implement the address busses and we would run out of PIC’s pins at the very beginning of application development. There has to be a better way and eProDas is proud to offer it at your exposal.

### 16.2.1 ADD_Port_BitMask

Prototypes in C/C++

```
int eProDas_ADD_Port_BitMask(unsigned int DeviceIdx, unsigned int Port,
  unsigned int StartPin, unsigned int EndPin, unsigned int AddValue,
  unsigned int MatchValue, unsigned int ResetValue);
```

Prototypes in Delphi

```
function eProDas_ADD_Port_BitMask(DeviceIdx: LongWord; Port: LongWord;
  StartPin: LongWord; EndPin: LongWord; AddValue: LongWord; MatchValue: LongWord;
  ResetValue: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_ADD_Port_BitMask(ByVal DeviceIdx As UInteger,
  ByVal Port As UInteger, ByVal StartPin As UInteger, ByVal EndPin As UInteger,
  ByVal AddValue As UInteger, ByVal MatchValue As UInteger,
  ByVal ResetValue As UInteger) As Integer
```

Like the already described function ADD_Port, this function adds an arbitrary number ADD_Value to the selected Port. However, contrary to the plain ADD_Port function this time the operation affects only a subset of adjacent pins that are specified by parameters StartPin and EndPin.

For example, to implement the previously described set of three independent address buses by using only port B, we could do the following. Let's choose that the "main" address bus occupies pins from 0 to 2 of port B. To increment it we would call the function ADD_Port_BitMask with the following parameters: Port=eProDas_Index_PORTB, StartPin=0, EndPin=2, AddValue=1.

Following the sequel, the "alternative" bus occupies pins 3 and 4 of port B and we increment it by calling the function ADD_Port_BitMask with the parameters: Port=eProDas_Index_PORTB, StartPin=3, EndPin=4, AddValue=8. Note that this time we do not add the value of 1 but we must instead add the weight of the start pin to achieve the increment functionality. Similarly, the "channel select" address bus is incremented by using the parameters: Port=eProDas_Index_PORTB, StartPin=5, EndPin=7, AddValue=32.

So far we have ignored two parameters MatchValue and ResetValue as you surely have noticed. These are needed in general cases where you want to address less peripheral units than the address bus is capable to swallow. Imagine that you connect only 6 peripheral units (addresses from 0 to 5) to the previously referenced 3-bit "main" address bus. Now, if you do a simple increment by the help of the function ADD_Port_BitMask, the first 6 bus cycles would be done correctly, but the seventh and the eighth ones would address some portion of the vacuum around your circuit. To adapt to such situations use the parameters MatchValue and ResetValue.

Under the hood the story goes like this. When the number ADD_Value is added to the current port's latch value, the temporal result (which is not written back to the port's latch yet) is subject to a bitmask where only the bits from StartPin to EndPin are preserved and all other pins are cleared. Then the value is compared to the MatchValue and if they are equal then the temporal result is reset to the ResetValue; both MatchValue and ResetValue are also subject to the same bitmask so you do not have to worry about the state of the irrelevant pins. Then the result is written back to the port's latch in such a way that bits outside the scope from StartPin to EndPin are preserved. No spikes or unnecessary transients are generated on external pins during this process.

Therefore, to implement the “main” bus that connects six peripheral units set `MatchValue` to 6 and `ResetValue` to zero. Generally, `ResetValue` should always be zero unless address 0 is reserved for “no periphery” as it was mentioned before. Again, `MatchValue` should reflect the actual bit pattern of the value that needs to be reset, so to implement the “channel select” bus with 6 temperature sensors, set `MatchValue` to 128+64 = 192. To exploit the full range of address bus (i.e. eight peripherals on a 3-bit bus) set both discussed values to zero. Then upon rollover, when the value reaches 0, it is going to be reset to 0 and address sequence will be unaltered by this sticking match/reset feature thing.

**Note.** The discussed function always works on the current latch value and it does nothing to initialize it. You have to make sure that the port’s latch value has an appropriate initial value by initializing it (say, by using the function `WritePort`) when your application starts or more generally somewhere prior to using the address bus features.

### 16.2.2 ROT_Port_BitMask

Prototypes in C/C++

```
int eProDas_ROT_Port_BitMask(unsigned int DeviceIdx, unsigned int Port,
  unsigned int StartPin, unsigned int EndPin);
```

Prototypes in Delphi

```
function eProDas_ROT_Port_BitMask(DeviceIdx: LongWord; Port: LongWord;
  StartPin: LongWord; EndPin: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_ROT_Port_BitMask(ByVal DeviceIdx As UInteger,
  ByVal Port As UInteger, ByVal StartPin As UInteger, ByVal EndPin As UInteger)
  As Integer
```

Incrementing address bus is the usual thing in the world of microprocessors. However, when connecting periphery to microcontrollers we often utilize the second kind of address bus, which is implemented by rotating bits of a port in question. This way we can save external decoder chip (like 74HC138) in the cases where we have spare PIC’s pins. Again, eProDas supports such activities.

To rotate a certain group of adjacent pins to the left and preserve all other pins intact use the function `ROT_Port_BitMask`. The parameters are self descriptive. With this function it is extremely vital to initialize the port’s latch value to a properly initial value since generally you want only one periphery to be activated at a time and you need to make sure that there is only one cleared bit (in the majority of cases *ChipSelect* pins are active in low state) in the rotating sequence.

For example, to address 5 peripheral units without consuming the external decoder you may reserve pins from 0 to 4 of port D for the task and initialize it to the binary value “xyz1 1110”. Then after executing the function `ROT_Port_BitMask` with parameters: `Port`=`eProDas_Index_PORTD`, `StartPin`=0, `EndPin`=4, the value would change to “xyz1 1101”. Additional calls of the function with the same parameters would result in the sequence “xyz1 1011”, “xyz1 0111”, “xyz0 1111”, “xyz1 1110”...

### 16.2.3 WritePort_BitMask

Prototypes in C/C++

```
int eProDas_WritePort_BitMask(unsigned int DeviceIdx, unsigned int Port,
  unsigned int StartPin, unsigned int EndPin, unsigned char Value);
```

Prototypes in Delphi

```
function eProDas_WritePort_BitMask(DeviceIdx: LongWord; Port: LongWord;
  StartPin: LongWord; EndPin: LongWord; Value: Byte):integer;
```

Prototypes in VisualBasic

```
Function eProDas_WritePort_BitMask(ByVal DeviceIdx As UInteger,
ByVal Port As UInteger, ByVal StartPin As UInteger, ByVal EndPin As UInteger,
ByVal Value As Byte) As Integer
```

This function writes the `Value` into the specified port, but it affects only the adjacent group of bits that are specified by `StartPin` and `EndPin`. With this function you can, for example, address devices on your complicated buses in a random access manner besides many more useful ways of usage.

### 16.2.4 *xxx*_BitMask_Toggle

Prototypes in C/C++

```
int eProDas_ADD_Port_BitMask_TogglePin(unsigned int DeviceIdx, unsigned int Port,
  unsigned int StartPin, unsigned int EndPin, unsigned int AddValue,
  unsigned int MatchValue, unsigned int ResetValue, unsigned int TogglePort,
  unsigned int TogglePin);

int eProDas_ROT_Port_BitMask_TogglePin(unsigned int DeviceIdx, unsigned int Port,
  unsigned int StartPin, unsigned int EndPin, unsigned int TogglePort,
  unsigned int TogglePin);

int eProDas_WritePort_BitMask_TogglePin(unsigned int DeviceIdx, unsigned int Port,
  unsigned int StartPin, unsigned int EndPin, unsigned char Value,
  unsigned int TogglePort, unsigned int TogglePin);
```

Prototypes in Delphi

```
function eProDas_ADD_Port_BitMask_TogglePin(DeviceIdx: LongWord; Port: LongWord;
  StartPin: LongWord; EndPin: LongWord; AddValue: LongWord; MatchValue: LongWord;
  ResetValue: LongWord; TogglePort: LongWord; TogglePin: LongWord):integer;

function eProDas_ROT_Port_BitMask_TogglePin(DeviceIdx: LongWord; Port: LongWord;
  StartPin: LongWord; EndPin: LongWord; TogglePort: LongWord;
  TogglePin: LongWord):integer;

function eProDas_WritePort_BitMask_TogglePin(DeviceIdx: LongWord; Port: LongWord;
  StartPin: LongWord; EndPin: LongWord; Value: Byte; TogglePort: LongWord;
  TogglePin: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_ADD_Port_BitMask_TogglePin(ByVal DeviceIdx As UInteger,
  ByVal Port As UInteger, ByVal StartPin As UInteger, ByVal EndPin As UInteger,
  ByVal AddValue As UInteger, ByVal MatchValue As UInteger,
  ByVal ResetValue As UInteger, ByVal TogglePort As UInteger,
  ByVal TogglePin As UInteger) As Integer

Function eProDas_ROT_Port_BitMask_TogglePin(ByVal DeviceIdx As UInteger,
  ByVal Port As UInteger, ByVal StartPin As UInteger, ByVal EndPin As UInteger,
  ByVal TogglePort As UInteger, ByVal TogglePin As UInteger) As Integer

Function eProDas_WritePort_BitMask_TogglePin(ByVal DeviceIdx As UInteger,
  ByVal Port As UInteger, ByVal StartPin As UInteger, ByVal EndPin As UInteger,
  ByVal Value As Byte, ByVal TogglePort As UInteger, ByVal TogglePin As UInteger)
  As Integer
```

When working with address buses you often utilize an additional control pin that signals when the valid address is presented on the address bus. This pin is sometimes referred to as *AddressStrobe*, *MemoryRequest* or something similar. The three functions that we are discussing now are aimed to help you achieve this functionality and they work in the same way as their respective counterparts `ADD_Port_BitMask`, `ROT_Port_BitMask` and `WritePort_BitMask` do, except that in addition they also toggle the state of one specified pin to mimic the described *AddressStrobe* functionality. When the bus cycle is over you need to toggle the pin back to the inactive state, say by using the function `TogglePin` (only if you need this semantics, otherwise eProDas does not demand from you to do it so, of course).

## 16.3 Operations on internal AD converter

These functions extend the operations on internal AD converter which are covered in section 15.2. Note that the AD module must be properly configured and enabled prior to calling these functions.

### 16.3.1 WaitInternalAD_*XXX*_Threshold

Prototype in C/C++

```
int eProDas_WaitInternalADAboveThreshold(unsigned int DeviceIdx,
  unsigned int Channel, unsigned int Threshold, unsigned int &Result);

int eProDas_WaitInternalADBelowThreshold(unsigned int DeviceIdx,
  unsigned int Channel, unsigned int Threshold, unsigned int &Result);

int eProDas_WaitInternalADBetweenThresholds(unsigned int DeviceIdx,
  unsigned int Channel, unsigned int LowThreshold,
  unsigned int HighThreshold, unsigned int &Result);
```

Prototype in Delphi

```
function eProDas_WaitInternalADAboveThreshold(DeviceIdx: LongWord;
  Channel: LongWord; Threshold: LongWord; var Result: LongWord):integer;

function eProDas_WaitInternalADBelowThreshold(DeviceIdx: LongWord;
  Channel: LongWord; Threshold: LongWord; var Result: LongWord):integer;

function eProDas_WaitInternalADBetweenThresholds(DeviceIdx: LongWord;
  Channel: LongWord; LowThreshold: LongWord; HighThreshold: LongWord;
  var Result: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_WaitInternalADAboveThreshold(ByVal DeviceIdx As UInteger,
  ByVal Channel As UInteger, ByVal Threshold As UInteger, ByRef Result As UInteger)
  As Integer

Function eProDas_WaitInternalADBelowThreshold(ByVal DeviceIdx As UInteger,
  ByVal Channel As UInteger, ByVal Threshold As UInteger, ByRef Result As UInteger)
  As Integer

Function eProDas_WaitInternalADBetweenThresholds(ByVal DeviceIdx As UInteger,
  ByVal Channel As UInteger, ByVal LowThreshold As UInteger,
  ByVal HighThreshold As UInteger, ByRef Result As UInteger) As Integer
```

Use the function `WaitInternalADAboveThreshold` to wait until some internal AD channel value is above certain threshold. Similarly, the function `WaitInternalADBelowThreshold` waits until the selected AD channel value is below the specified threshold. Finally, with the function `WaitInternalADBetweenThresholds` you can wait until certain internal AD channel value is between two thresholds.

The valid threshold values are between 1 and 1022. The equality of AD channel value with the threshold is always considered a fulfilment of the threshold condition. The result of AD conversion that fulfils the condition is returned in the `Result` parameter. The eProDas device is frozen forever if the condition is cannot be fulfilled.

### 16.3.2 WaitInternalADTransition

Prototype in C/C++

```
int eProDas_WaitInternalADTransition(unsigned int DeviceIdx, unsigned int Channel,
  unsigned int Threshold, unsigned int LowToHigh, unsigned int &Result);
```

Prototype in Delphi

```
function eProDas_WaitInternalADTransition(DeviceIdx: LongWord; Channel: LongWord;
  Threshold: LongWord; LowToHigh: LongWord; var Result: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_WaitInternalADTransition(ByVal DeviceIdx As UInteger,
  ByVal Channel As UInteger, ByVal Threshold As UInteger,
  ByVal LowToHigh As UInteger, ByRef Result As UInteger) As Integer
```

Instead of merely waiting for a state of AD input you can wait on its crossing of a prescribed threshold. If the parameter `LowToHigh` is non-zero, then the waiting period lasts till AD input is first detected below the threshold and then it rises above it; in the opposite case AD input must be first spotted above the threshold and then lowered below it. In both cases the final AD result that fulfils the condition is returned through the parameter `Result`.

## 16.4 Operations on internal comparators

These functions extend the operations on internal comparators which are covered in section 15.3. Note that comparators must be properly configured and enabled prior to calling these functions.

### 16.4.1 QueryInternalComparatorsChange

Prototype in C/C++

```
int eProDas_QueryInternalComparatorsChange(unsigned int DeviceIdx,
  unsigned int &Changed, unsigned int &Result);
```

Prototype in Delphi

```
function eProDas_QueryInternalComparatorsChange(DeviceIdx: LongWord;
  var Changed: LongWord; var Result: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_QueryInternalComparatorsChange(ByVal DeviceIdx As UInteger,
  ByRef Changed As UInteger, ByRef Result As UInteger) As Integer
```

This function not only queries the state of comparators but it also reads the indicator (comparator interrupt flag) of state change, which is set if any or both active comparator's outputs change.

If the change occurred after the last call to this function the return parameter `Changed` is set to a non-zero value and the current state of the comparators is returned in the two LSBs of the `Result` as it is the case with the function `ReadInternalComparators` (section 15.3.2). However, if the state of comparators did not change, then the value of `Result` is set to zero regardless of the current state of comparators. This peculiarity is due to the warning in the PIC's datasheet about the possibility of incorrectly updated comparators state if the change occurs in the moment of reading the comparators register. To eliminate this danger we simply do not check the current state if indicator does not reveals the change.

**Note 1.** The indicator remains set until queried even if comparator(s) return to the initial state after a change.  Therefore, you do not need to be concerned that you will miss the change if you application does not poll the indicator often enough.

By reading the indicator we cannot tell which comparator changed so the user application has to remember the previous state and make a comparison with the new one to establish the exact nature of event (not necessary if you only use one comparator, of course; but remember that unless comparators are in mode 1 (please see the Figure 37), the unused one must be connected in such a way that its output cannot change or this simplification will not work). Also, if after the change, the state of comparators returns to the initial one (i.e. if the signal(s) on input pin(s) returns into the initial voltage area) before you read the new state, you will not be able to know what happened and which comparator experienced the change of it's state.

The indicator of change is always cleared by this function so calling it twice in a short time (under the presumption that the voltages on comparator inputs vary slowly), the second call will always return zero value of parameter `Changed`.

**Note 2.** Always call this function at the beginning of usage of comparators to clear the indicator of change and start afresh. The indicator may be falsely set when comparators are switched on.

**Note 3.** According to PIC's datasheet there is a possibility that the indicator is falsely set upon change of mode of comparators operation. Therefore, always set comparators mode first, then call this function to clear indicator of changes and then start to observe the true changes of comparator outputs.

### 16.4.2 WaitInternalComparatorsState

Prototype in C/C++

```
int eProDas_WaitInternalComparatorsState(unsigned int DeviceIdx,
  unsigned int MonitorComp1, unsigned int MonitorComp2, unsigned int State1,
  unsigned int State2);
```

Prototype in Delphi

```
function eProDas_WaitInternalComparatorsState(DeviceIdx: LongWord;
  MonitorComp1: LongWord; MonitorComp2: LongWord; State1: LongWord;
  State2: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_WaitInternalComparatorsState(ByVal DeviceIdx As UInteger,
  ByVal MonitorComp1 As UInteger, ByVal MonitorComp2 As UInteger,
  ByVal State1 As UInteger, ByVal State2 As UInteger) As Integer
```

Use this function to wait for a prescribed state of one or both comparators. Non-zero value of parameters `MonitorComp1` and `MonitorComp2` indicates that the respective comparators 1 and 2 are monitored; otherwise the respective comparator is ignored. Note that at least one comparator must be monitored or the function returns with the error `eProDas_Error_InvalidComparatorMode`.

The prescribed state of the comparators is specified by the parameter `State1` and `State2`, where a non-zero value demands high logic state of the respective comparator's output; for the ignored comparator the state value is also ignored.

The function returns when all non-ignored comparators are in a prescribed state. If such condition never arises, the eProDas device will patiently wait forever and only power outage may make it change its mind. Note that the comparator's module must be properly configured prior to calling this function.

### 16.4.3 RepetitivelyWaitInternalComparatorsState

Prototype in C/C++

```
int eProDas_RepetitivelyWaitInternalComparatorsState(unsigned int DeviceIdx,
  unsigned int Repetitions, unsigned int MonitorComp1, unsigned int MonitorComp2,
  unsigned int State1, unsigned int State2);
```

Prototype in Delphi

```
function eProDas_RepetitivelyWaitInternalComparatorsState(DeviceIdx: LongWord;
  Repetitions: LongWord; MonitorComp1: LongWord; MonitorComp2: LongWord;
  State1: LongWord; State2: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_RepetitivelyWaitInternalComparatorsState
   (ByVal DeviceIdx As UInteger, ByVal Repetitions As UInteger,
  ByVal MonitorComp1 As UInteger, ByVal MonitorComp2 As UInteger,
  ByVal State1 As UInteger, ByVal State2 As UInteger) As Integer
```

A variation of the previous function where the state must be read for `Repetitions` times (from 1 to 65 535) successfully in a row in order to be recognized as stable. Whenever the read state differs from the specified one, the counting starts from the beginning. The adjacent readings are 500 ns apart.

**Remark.** In practical realizations of electronic systems we often use Schmitt triggers instead of plain comparators to filter out noise and spikes, which would trigger false state transitions; in the case of counting such transition events, the consequences are disastrous. Schmitt trigger solves the issue by providing a hysteresis of state-changing threshold. Now, repetitive reading of comparator's state may be viewed as implementation of a self adjusting dynamic Schmitt trigger in software. Namely, if the demanded number of unchanged repetitions is fairly high the state of analog signal that is near the comparator's threshold will never be declared stable due to inevitable spurious threshold transitions due to noise. More noise in the signal would require a signal level farther away from the threshold to be declared stable by means of which the hysteresis is self adjusting. Isn't the engineering work great?

**Note.** The solution in the previous paragraph may not be a suitable replacement for Schmitt trigger in all situations. Please, make sure that you know what you are doing before you start sending us black magic curses.

### 16.4.4 WaitStableInternalComparatorsState

Prototype in C/C++

```
int eProDas_WaitStableInternalComparatorsState(unsigned int DeviceIdx,
  unsigned int Repetitions, unsigned int MonitorComp1, unsigned int MonitorComp2,
  unsigned char &Result);
```

Prototype in Delphi

```
function eProDas_WaitStableInternalComparatorsState(DeviceIdx: LongWord;
  Repetitions: LongWord; MonitorComp1: LongWord; MonitorComp2: LongWord;
  var Result: Byte):integer;
```

Prototype in VisualBasic

```
Function eProDas_WaitStableInternalComparatorsState(ByVal DeviceIdx As UInteger,
  ByVal Repetitions As UInteger, ByVal MonitorComp1 As UInteger,
  ByVal MonitorComp2 As UInteger, ByRef Result As Byte) As Integer
```

A variation of the previous function, where the state is not prescribed in advance. The only thing that matters is that the state does not change during the `Repetitions` consecutive readings. The state that fulfils this condition is returned in the parameter `Result`, where bits 0 and 1 carry the state of comparators 1 and 2, respectively.

### 16.4.5 WaitInternalComparatorTransition

Prototype in C/C++

```
int eProDas_WaitInternalComparatorTransition(unsigned int DeviceIdx,
  unsigned int Comparator, unsigned int LowToHigh);
```

Prototype in Delphi

```
function eProDas_WaitInternalComparatorTransition(DeviceIdx: LongWord;
  Comparator: LongWord; LowToHigh: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_WaitInternalComparatorTransition(ByVal DeviceIdx As UInteger,
  ByVal Comparator As UInteger, ByVal LowToHigh As UInteger) As Integer
```

Instead of waiting for a certain state you may need to wait for a transition of one of the comparators; a non-zero value of `Comparator` specifies comparator 2, otherwise its comparator's 1 game.

If the parameter `LowToHigh` is non-zero the function waits until the selected comparator is in low state and then changes to the high one, otherwise the sequence is reversed.

### 16.4.6 RepetitivelyWaitInternalComparatorTransition

Prototype in C/C++

```
int eProDas_RepetitivelyWaitInternalComparatorTransition(unsigned int DeviceIdx,
  unsigned int Repetitions, unsigned int Comparator, unsigned int LowToHigh);
```

Prototype in Delphi

```
function eProDas_RepetitivelyWaitInternalComparatorTransition(DeviceIdx: LongWord;
  Repetitions: LongWord; Comparator: LongWord; LowToHigh: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_RepetitivelyWaitInternalComparatorTransition
  (ByVal DeviceIdx As UInteger, ByVal Repetitions As UInteger,
  ByVal Comparator As UInteger, ByVal LowToHigh As UInteger) As Integer
```

And a variation of the previous function, where both states are monitored repetitively for `Repetitions` number of times before each state is recognized as stable, by means of which noise and spikes are filtered out.

## 16.5 Extended set of SPI related functions

Functions in this section extend the functionality of eProDas functions for transferring data over SPI bus (section 15.6.5) by means of covering many real-world SPI situations, like handling of *ChipSelect* pin, triggering *LoadStrobe* pin at the end of the transfer and waiting for *DataReady* signal before exchanging data.

All these tasks and more could easily be accomplished by using a combination of just one function for doing SPI transfer and the long time ago described functions for working with digital ports (sections 15.1 and 16.1). For example, *ChipSelect* signal could be handled by calling the function `TogglePin` prior initiating the transfer, then data would be exchanged over SPI bus, and finally we would toggle the *ChipSelect* pin one more time.

Although such scheme works it is rather inefficient. The problem is that we need to send three isolated USB packets from host PC to eProDas to accomplish the exemplar task. As you have learned in section 14.6.1.1, sending of large number of small USB packets is much slower than sending one large packet. For that matter eProDas functions in this section try to encapsulate all SPI related actions that are needed to complete the entire SPI transfer cycle into one USB packet.

If your transfer scheme does not fit into any of the next-to-be presented functions, do not panic. Under the hood, all of them rely on a common eProDas `Transfer` backbone, which is in fact a simple interpreter of transfer oriented commands that are contained in one USB packet. By writing your chunks of code in this interpreted “language” you can deal with a lot of situations that we did not come up with. Please, see the somewhat hard to read section 23.2 if you are interested in the details.

### 16.5.1 MasterSPI_TransferByteCS group of functions

Prototypes in C/C++

```
int eProDas_MasterSPI_TransferByteCS(unsigned int DeviceIdx,
  unsigned char WriteByte, unsigned char &ReadByte, unsigned int ChipSelectPort,
  unsigned int ChipSelectBit);

int eProDas_MasterSPI_TransferByteOUT_CS(unsigned int DeviceIdx,
  unsigned char WriteByte, unsigned int ChipSelectPort,
  unsigned int ChipSelectBit);

int eProDas_MasterSPI_TransferByteIN_CS(unsigned int DeviceIdx,
  unsigned char &ReadByte, unsigned int ChipSelectPort,
  unsigned int ChipSelectBit);
```

Prototypes in Delphi

```
function eProDas_MasterSPI_TransferByteCS(DeviceIdx: LongWord; WriteByte: Byte;
  var ReadByte: Byte; ChipSelectPort: LongWord; ChipSelectBit: LongWord):integer;

function eProDas_MasterSPI_TransferByteOUT_CS(DeviceIdx: LongWord; WriteByte: Byte;
  var ReadByte: Byte; ChipSelectPort: LongWord; ChipSelectBit: LongWord):integer;

function eProDas_MasterSPI_TransferByteIN_CS(DeviceIdx: LongWord; WriteByte: Byte;
  var ReadByte: Byte; ChipSelectPort: LongWord; ChipSelectBit: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_MasterSPI_TransferByteCS(ByVal DeviceIdx As UInteger,
  ByVal WriteByte As Byte, ByRef ReadByte As Byte,
  ByVal ChipSelectPort As UInteger, ByVal ChipSelectBit As UInteger) As Integer

Function eProDas_MasterSPI_TransferByteOUT_CS(ByVal DeviceIdx As UInteger,
  ByVal WriteByte As Byte, ByVal ChipSelectPort As UInteger,
  ByVal ChipSelectBit As UInteger) As Integer

Function eProDas_MasterSPI_TransferByteIN_CS(ByVal DeviceIdx As UInteger,
  ByRef ReadByte As Byte, ByVal ChipSelectPort As UInteger,
  ByVal ChipSelectBit As UInteger) As Integer
```

Generally, SPI periphery requires proper handling of *ChipSelect* or some other similar signal in order to function properly, which is not done automatically by the functions `MasterSPI_TransferByte`*`XXX`* (section 15.6.5.1), since this handling depends on the situation (some times this is unneeded or you might decide to implement a more or less complicated address bus, like section 16.2 discusses). In the simple cases, where *ChipSelect* is directly connected to some PIC’s pin, the functions in this section may help you accomplish the task.

These functions work precisely in the same way as the respective functions from the group `MasterSPI_TransferByte` do, except that now they toggle the state of an arbitrary specified pin prior initiating the transfer and also after the completion of operation. The pin to be toggled is specified in the same way as when working with the function `SetInternalSPI_MasterMode_CS` (including the skipping of pin toggling when `ChipSelectPort` is set to 1 000 or more).

**Note.** Prior to calling any of these functions make sure that the pin in question is configured as digital output or no toggling of its state can be done. Also, the state of the pin must be in *ChipSelect* inactive state (usually logic 1) prior to calling the function for this scenario to work properly.

### 16.5.2 MasterSPI_TransferByteTG group of functions

Prototypes in C/C++

```
int eProDas_MasterSPI_TransferByteTG(unsigned int DeviceIdx,
  unsigned char WriteByte, unsigned char &ReadByte, unsigned int ChipSelectPort,
  unsigned int ChipSelectBit, unsigned int TogglePort, unsigned int ToggleBit);

int eProDas_MasterSPI_TransferByteOUT_TG(unsigned int DeviceIdx,
  unsigned char WriteByte, unsigned int ChipSelectPort,
  unsigned int ChipSelectBit, unsigned int TogglePort, unsigned int ToggleBit);

int eProDas_MasterSPI_TransferByteIN_TG(unsigned int DeviceIdx,
  unsigned char &ReadByte, unsigned int ChipSelectPort,
  unsigned int ChipSelectBit, unsigned int TogglePort, unsigned int ToggleBit);
```

Prototypes in Delphi

```
function eProDas_MasterSPI_TransferByteTG(DeviceIdx: LongWord; WriteByte: Byte;
  var ReadByte: Byte; ChipSelectPort: LongWord; ChipSelectBit: LongWord;
  TogglePort: LongWord; ToggleBit: LongWord):integer;

function eProDas_MasterSPI_TransferByteOUT_TG(DeviceIdx: LongWord; WriteByte: Byte;
  var ReadByte: Byte; ChipSelectPort: LongWord; ChipSelectBit: LongWord;
  TogglePort: LongWord; ToggleBit: LongWord):integer;

function eProDas_MasterSPI_TransferByteIN_TG(DeviceIdx: LongWord; WriteByte: Byte;
  var ReadByte: Byte; ChipSelectPort: LongWord; ChipSelectBit: LongWord;
  TogglePort: LongWord; ToggleBit: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_MasterSPI_TransferByteTG(ByVal DeviceIdx As UInteger,
  ByVal WriteByte As Byte, ByRef ReadByte As Byte,
  ByVal ChipSelectPort As  UInteger, ByVal ChipSelectBit As UInteger,
  ByVal TogglePort As UInteger, ByVal ToggleBit As UInteger) As Integer

Function eProDas_MasterSPI_TransferByteOUT_TG(ByVal DeviceIdx As UInteger,
  ByVal WriteByte As Byte, ByVal ChipSelectPort As UInteger,
  ByVal ChipSelectBit As UInteger, ByVal TogglePort As UInteger,
  ByVal ToggleBit As UInteger) As Integer

Function eProDas_MasterSPI_TransferByteIN_TG(ByVal DeviceIdx As UInteger,
  ByRef ReadByte As Byte, ByVal ChipSelectPort As UInteger,
  ByVal ChipSelectBit As UInteger, ByVal TogglePort As UInteger,
  ByVal ToggleBit As UInteger) As Integer
```

These functions combine the functionality of their respective functions from the previous group with additional double toggling of the arbitrary pin after the transfer is completed. The pin is specified by the last two function parameters in a usual way. The behaviour supports e.g. some DA integrated circuits that double buffer the DA register and reflect the actual written value on an analog output pin only when transition on a certain pin (like *LoadStrobe*) is induced. Note that *ChipSelect* is returned to inactive state **after** the double toggling of the pin and not before it.

### 16.5.3 MasterSPI_TransferByteWS group of functions

Prototypes in C/C++

```
int eProDas_MasterSPI_TransferByteWS(unsigned int DeviceIdx,
  unsigned char WriteByte, unsigned char &ReadByte, unsigned int ChipSelectPort,
  unsigned int ChipSelectBit, unsigned int WaitPort, unsigned int WaitPin,
  unsigned int WaitState);

int eProDas_MasterSPI_TransferByteOUT_WS(unsigned int DeviceIdx,
  unsigned char WriteByte, unsigned int ChipSelectPort, unsigned int ChipSelectBit,
  unsigned int WaitPort, unsigned int WaitPin, unsigned int WaitState);

int eProDas_MasterSPI_TransferByteIN_WS(unsigned int DeviceIdx,
  unsigned char &ReadByte, unsigned int ChipSelectPort, unsigned int ChipSelectBit,
  unsigned int WaitPort, unsigned int WaitPin, unsigned int WaitState);
```

Prototypes in Delphi

```
function eProDas_MasterSPI_TransferByteWS(DeviceIdx: LongWord; WriteByte: Byte;
  var ReadByte: Byte; ChipSelectPort: LongWord; ChipSelectBit: LongWord;
  WaitPort: LongWord; WaitPin: LongWord;
  WaitState: LongWord):integer;

function eProDas_MasterSPI_TransferByteOUT_WS(DeviceIdx: LongWord; WriteByte: Byte;
  ChipSelectPort: LongWord; ChipSelectBit: LongWord; WaitPort: LongWord;
  WaitPin: LongWord; WaitState: LongWord):integer;

function eProDas_MasterSPI_TransferByteIN_WS(DeviceIdx: LongWord;
  var ReadByte: Byte; ChipSelectPort: LongWord; ChipSelectBit: LongWord;
  WaitPort: LongWord; WaitPin: LongWord; WaitState: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_MasterSPI_TransferByteWS(ByVal DeviceIdx As UInteger,
  ByVal WriteByte As Byte, ByRef ReadByte As Byte,
  ByVal ChipSelectPort As   UInteger, ByVal ChipSelectBit As UInteger,
  ByVal WaitPort As UInteger, ByVal WaitPin As UInteger,
  ByVal WaitState As UInteger) As Integer

Function eProDas_MasterSPI_TransferByteOUT_WS(ByVal DeviceIdx As UInteger,
  ByVal WriteByte As Byte, ByVal ChipSelectPort As UInteger,
  ByVal ChipSelectBit As UInteger, ByVal WaitPort As UInteger,
  ByVal WaitPin As UInteger, ByVal WaitState As UInteger) As Integer

Function eProDas_MasterSPI_TransferByteIN_WS(ByVal DeviceIdx As UInteger,
  ByRef ReadByte As Byte, ByVal ChipSelectPort As UInteger,
  ByVal ChipSelectBit As UInteger, ByVal WaitPort As UInteger,
  ByVal WaitPin As UInteger, ByVal WaitState As UInteger) As Integer
```

Here comes another set of functions that are intended to support specific behaviour of some SPI chips that signal readiness of new data by toggling one of their pin. An example is an AD converter that automatically performs AD conversion periodically and signals to the consumer each time it does it so.

Therefore, these functions work in the same manner as the respective functions from the group `TransferByteCS` do, except that this time we wait for a certain state or transition on a selected pin (`WaitPort` and `WaitPin` combination of parameters) before SPI transfer starts. The parameter `WaitState` specifies what we are waiting for, according to the following table.

| `WaitState` code | named constant | wait for condition |
|---|---|---|
| 0 | `eProDas_WaitPin_Low` | low state on the pin |
| 1 | `eProDas_WaitPin_High` | high state on the pin |
| 2 | `eProDas_WaitPin_LowToHigh` | transition from low to high |
| 3 | `eProDas_WaitPin_HighToLow` | transition from high to low |

**Table 8: `WaitState` codes and their meaning.**

**Note.** *ChipSelect* is activated **after** the state is reached and not before that. If such behaviour is not suitable in your case (for example, your logic prevents `WaitPin` state to reach PIC without activated *ChipSelect*), use the next group of functions.

### 16.5.4 MasterSPI_TransferByteWS2 group of functions

Prototypes in C/C++

```
int eProDas_MasterSPI_TransferByteWS2(unsigned int DeviceIdx,
  unsigned char WriteByte, unsigned char &ReadByte, unsigned int ChipSelectPort,
  unsigned int ChipSelectBit, unsigned int WaitPort, unsigned int WaitPin,
  unsigned int WaitState);

int eProDas_MasterSPI_TransferByteOUT_WS2(unsigned int DeviceIdx,
  unsigned char WriteByte, unsigned int ChipSelectPort, unsigned int ChipSelectBit,
  unsigned int WaitPort, unsigned int WaitPin, unsigned int WaitState);

int eProDas_MasterSPI_TransferByteIN_WS2(unsigned int DeviceIdx,
  unsigned char &ReadByte, unsigned int ChipSelectPort, unsigned int ChipSelectBit,
  unsigned int WaitPort, unsigned int WaitPin, unsigned int WaitState);
```

Prototypes in Delphi

```
function eProDas_MasterSPI_TransferByteWS2(DeviceIdx: LongWord; WriteByte: Byte;
  var ReadByte: Byte; ChipSelectPort: LongWord; ChipSelectBit: LongWord;
  WaitPort: LongWord; WaitPin: LongWord;
  WaitState: LongWord):integer;

function eProDas_MasterSPI_TransferByteOUT_WS2(DeviceIdx: LongWord;
  WriteByte: Byte; ChipSelectPort: LongWord; ChipSelectBit: LongWord;
  WaitPort: LongWord; WaitPin: LongWord; WaitState: LongWord):integer;

function eProDas_MasterSPI_TransferByteIN_WS2(DeviceIdx: LongWord;
  var ReadByte: Byte; ChipSelectPort: LongWord; ChipSelectBit: LongWord;
  WaitPort: LongWord; WaitPin: LongWord; WaitState: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_MasterSPI_TransferByteWS2(ByVal DeviceIdx As UInteger,
  ByVal WriteByte As Byte, ByRef ReadByte As Byte,
  ByVal ChipSelectPort As  UInteger, ByVal ChipSelectBit As UInteger,
  ByVal WaitPort As UInteger, ByVal WaitPin As UInteger,
  ByVal WaitState As UInteger) As Integer

Function eProDas_MasterSPI_TransferByteOUT_WS2(ByVal DeviceIdx As UInteger,
  ByVal WriteByte As Byte, ByVal ChipSelectPort As UInteger,
  ByVal ChipSelectBit As UInteger, ByVal WaitPort As UInteger,
  ByVal WaitPin As UInteger, ByVal WaitState As UInteger) As Integer

Function eProDas_MasterSPI_TransferByteIN_WS2(ByVal DeviceIdx As UInteger,
  ByRef ReadByte As Byte, ByVal ChipSelectPort As UInteger,
  ByVal ChipSelectBit As UInteger, ByVal WaitPort As UInteger,
  ByVal WaitPin As UInteger, ByVal WaitState As UInteger) As Integer
```

These functions work in a precisely the same way as the previously described functions do, except that *ChipSelect* is activated before waiting on a pin state and not other way round. Such behaviour is needed if your (address) decoding logic must activate the chip to let the waited for signal pass through external latches etc. to reach the PIC.

### 16.5.5 MasterSPI_TransferArrayCS group of functions

Prototypes in C/C++

```
eProDasDLL_API int eProDas_MasterSPI_TransferArrayCS(unsigned int DeviceIdx,
  unsigned char *WriteArray, unsigned char *ReadArray, unsigned int Length,
  unsigned int ChipSelectPort, unsigned int ChipSelectBit);

eProDasDLL_API int eProDas_MasterSPI_TransferArrayOUT_CS(unsigned int DeviceIdx,
  unsigned char *WriteArray, unsigned int Length, unsigned int ChipSelectPort,
  unsigned int ChipSelectBit);

eProDasDLL_API int eProDas_MasterSPI_TransferArrayIN_CS(unsigned int DeviceIdx,
  unsigned char *ReadArray, unsigned int Length, unsigned int ChipSelectPort,
  unsigned int ChipSelectBit);
```

Prototypes in Delphi

```
function eProDas_MasterSPI_TransferArrayCS(DeviceIdx: LongWord; WriteArray: PByte;
  ReadArray: PByte; Length: LongWord; ChipSelectPort: LongWord;
  ChipSelectBit: LongWord):integer;

function eProDas_MasterSPI_TransferArrayOUT_CS(DeviceIdx: LongWord;
  WriteArray: PByte; Length: LongWord; ChipSelectPort: LongWord;
  ChipSelectBit: LongWord):integer;

function eProDas_MasterSPI_TransferArrayIN_CS(DeviceIdx: LongWord;
  ReadArray: PByte; Length: LongWord; ChipSelectPort: LongWord;
  ChipSelectBit: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_MasterSPI_TransferArrayCS(ByVal DeviceIdx As UInteger,
  ByRef WriteArray As Byte, ByRef ReadArray As Byte, ByVal Length As UInteger,
  ByVal ChipSelectPort As UInteger, ByVal ChipSelectBit As UInteger) As Integer

Function eProDas_MasterSPI_TransferArrayOUT_CS(ByVal DeviceIdx As UInteger,
  ByRef WriteArray As Byte, ByVal Length As UInteger,
  ByVal ChipSelectPort As UInteger, ByVal ChipSelectBit As UInteger) As Integer

Function eProDas_MasterSPI_TransferArrayIN_CS(ByVal DeviceIdx As UInteger,
  ByRef ReadArray As Byte, ByVal Length As UInteger,
  ByVal ChipSelectPort As UInteger, ByVal ChipSelectBit As UInteger) As Integer
```

These functions combine the functionality of the respective functions within the group `MasterSPI_TransferArray` (section 15.6.5.2) with the handling of peripheral *ChipSelect* signal as the functions `MasterSPI_TransferByteCS` do. The maximal transfer length is 37 bytes.

**Note.** These functions are intended to transfer multi-byte data that constitute one peripheral bus cycle. Therefore, the signal *ChipSelect* is activated only before transferring the first byte and is deactivated upon completion of the last byte transfer. This functionality supports the working of majority of AD and DA chips, where for example 24-bit AD result is transferred to PIC in three 8-bit chunks within one AD active cycle.

### 16.5.6 MasterSPI_TransferArrayTG group of functions

Prototypes in C/C++

```
eProDasDLL_API int eProDas_MasterSPI_TransferArrayTG(unsigned int DeviceIdx,
  unsigned char *WriteArray, unsigned char *ReadArray, unsigned int Length,
  unsigned int ChipSelectPort, unsigned int ChipSelectBit, unsigned int TogglePort,
  unsigned int ToggleBit);

eProDasDLL_API int eProDas_MasterSPI_TransferArrayOUT_TG(unsigned int DeviceIdx,
  unsigned char *WriteArray, unsigned int Length, unsigned int ChipSelectPort,
  unsigned int ChipSelectBit, unsigned int TogglePort, unsigned int ToggleBit);

eProDasDLL_API int eProDas_MasterSPI_TransferArrayIN_TG(unsigned int DeviceIdx,
  unsigned char *ReadArray, unsigned int Length, unsigned int ChipSelectPort,
  unsigned int ChipSelectBit, unsigned int TogglePort, unsigned int ToggleBit);
```

Prototypes in Delphi

```
function eProDas_MasterSPI_TransferArrayTG(DeviceIdx: LongWord; WriteArray: PByte;
  ReadArray: PByte; Length: LongWord; ChipSelectPort: LongWord;
  ChipSelectBit: LongWord; TogglePort: LongWord; ToggleBit: LongWord):integer;

function eProDas_MasterSPI_TransferArrayOUT_TG(DeviceIdx: LongWord;
  WriteArray: PByte; Length: LongWord; ChipSelectPort: LongWord;
  ChipSelectBit: LongWord; TogglePort: LongWord; ToggleBit: LongWord):integer;

function eProDas_MasterSPI_TransferArrayIN_TG(DeviceIdx: LongWord;
  ReadArray: PByte; Length: LongWord; ChipSelectPort: LongWord;
  ChipSelectBit: LongWord; TogglePort: LongWord; ToggleBit: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_MasterSPI_TransferArrayTG(ByVal DeviceIdx As UInteger,
  ByRef WriteArray As Byte, ByRef ReadArray As Byte, ByVal Length As UInteger,
  ByVal ChipSelectPort As UInteger, ByVal ChipSelectBit As UInteger,
  ByVal TogglePort As UInteger, ByVal ToggleBit As UInteger) As Integer

Function eProDas_MasterSPI_TransferArrayOUT_TG(ByVal DeviceIdx As UInteger,
  ByRef WriteArray As Byte, ByVal Length As UInteger,
  ByVal ChipSelectPort As UInteger, ByVal ChipSelectBit As UInteger,
  ByVal TogglePort As UInteger, ByVal ToggleBit As UInteger) As Integer

Function eProDas_MasterSPI_TransferArrayIN_TG(ByVal DeviceIdx As UInteger,
  ByRef ReadArray As Byte, ByVal Length As UInteger,
  ByVal ChipSelectPort As UInteger, ByVal ChipSelectBit As UInteger,
  ByVal TogglePort As UInteger, ByVal ToggleBit As UInteger) As Integer
```

Analogous to the `TransferByteTG` these functions combine the functionality of their respective functions from the previous group with additional double toggling of an arbitrary pin after the transfer of the last byte is completed but before *ChipSelect* is deactivated. The pin is specified by two the last parameters in the usual way. The maximal transfer length is 35 bytes.

### 16.5.7 MasterSPI_TransferArrayWS group of functions

Prototypes in C/C++

```
int eProDas_MasterSPI_TransferArrayWS(unsigned int DeviceIdx,
  unsigned char *WriteArray, unsigned char *ReadArray, unsigned int Length,
  unsigned int ChipSelectPort, unsigned int ChipSelectBit, unsigned int WaitPort,
  unsigned int WaitPin, unsigned int WaitState);

int eProDas_MasterSPI_TransferArrayOUT_WS(unsigned int DeviceIdx,
  unsigned char *WriteArray, unsigned int Length, unsigned int ChipSelectPort,
  unsigned int ChipSelectBit, unsigned int WaitPort, unsigned int WaitPin,
  unsigned int WaitState);

int eProDas_MasterSPI_TransferArrayIN_WS(unsigned int DeviceIdx,
  unsigned char *ReadArray, unsigned int Length, unsigned int ChipSelectPort,
  unsigned int ChipSelectBit, unsigned int WaitPort, unsigned int WaitPin,
  unsigned int WaitState);
```

Prototypes in Delphi

```
function eProDas_MasterSPI_TransferArrayWS(DeviceIdx: LongWord; WriteArray: PByte;
  ReadArray: PByte; Length: LongWord; ChipSelectPort: LongWord;
  ChipSelectBit: LongWord; WaitPort: LongWord; WaitPin: LongWord;
  WaitState: LongWord):integer;

function eProDas_MasterSPI_TransferArrayOUT_WS(DeviceIdx: LongWord;
  WriteArray: PByte; Length: LongWord; ChipSelectPort: LongWord;
  ChipSelectBit: LongWord; WaitPort: LongWord; WaitPin: LongWord;
  WaitState: LongWord):integer;

function eProDas_MasterSPI_TransferArrayIN_WS(DeviceIdx: LongWord;
  ReadArray: PByte; Length: LongWord; ChipSelectPort: LongWord;
  ChipSelectBit: LongWord; WaitPort: LongWord; WaitPin: LongWord;
  WaitState: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_MasterSPI_TransferArrayWS(ByVal DeviceIdx As UInteger,
  ByRef WriteArray As Byte, ByRef ReadArray As UInteger, ByVal Length As UInteger,
  ByVal ChipSelectPort As UInteger, ByVal ChipSelectBit As UInteger,
  ByVal WaitPort As UInteger, ByVal WaitPin As UInteger,
  ByVal WaitState As UInteger) As Integer

Function eProDas_MasterSPI_TransferArrayOUT_WS(ByVal DeviceIdx As UInteger,
  ByRef WriteArray As Byte, ByVal Length As UInteger,
  ByVal ChipSelectPort As UInteger, ByVal ChipSelectBit As UInteger,
  ByVal WaitPort As UInteger, ByVal WaitPin As UInteger,
  ByVal WaitState As UInteger) As Integer

Function eProDas_MasterSPI_TransferArrayIN_WS(ByVal DeviceIdx As UInteger,
  ByRef ReadArray As Byte, ByVal Length As UInteger,
  ByVal ChipSelectPort As UInteger, ByVal ChipSelectBit As UInteger,
  ByVal WaitPort As UInteger, ByVal WaitPin As UInteger,
  ByVal WaitState As UInteger) As Integer
```

These functions parallel the `TransferByteWS` group of functions. Their usage and working should be obvious from the knowledge of related functions and the above definitions. The maximal transfer length is 35 bytes.

### 16.5.8 MasterSPI_TransferArrayWS2 group of functions

Prototypes in C/C++

```
int eProDas_MasterSPI_TransferArrayWS2(unsigned int DeviceIdx,
  unsigned char *WriteArray, unsigned char *ReadArray, unsigned int Length,
  unsigned int ChipSelectPort, unsigned int ChipSelectBit, unsigned int WaitPort,
  unsigned int WaitPin, unsigned int WaitState);

int eProDas_MasterSPI_TransferArrayOUT_WS2(unsigned int DeviceIdx,
  unsigned char *WriteArray, unsigned int Length, unsigned int ChipSelectPort,
  unsigned int ChipSelectBit, unsigned int WaitPort, unsigned int WaitPin,
  unsigned int WaitState);

int eProDas_MasterSPI_TransferArrayIN_WS2(unsigned int DeviceIdx,
  unsigned char *ReadArray, unsigned int Length, unsigned int ChipSelectPort,
  unsigned int ChipSelectBit, unsigned int WaitPort, unsigned int WaitPin,
  unsigned int WaitState);
```

Prototypes in Delphi

```
function eProDas_MasterSPI_TransferArrayWS2(DeviceIdx: LongWord; WriteArray: PByte;
  ReadArray: PByte; Length: LongWord; ChipSelectPort: LongWord;
  ChipSelectBit: LongWord; WaitPort: LongWord; WaitPin: LongWord;
  WaitState: LongWord):integer;

function eProDas_MasterSPI_TransferArrayOUT_WS2(DeviceIdx: LongWord;
  WriteArray: PByte; Length: LongWord; ChipSelectPort: LongWord;
  ChipSelectBit: LongWord; WaitPort: LongWord; WaitPin: LongWord;
  WaitState: LongWord):integer;

function eProDas_MasterSPI_TransferArrayIN_WS2(DeviceIdx: LongWord;
  ReadArray: PByte; Length: LongWord; ChipSelectPort: LongWord;
  ChipSelectBit: LongWord; WaitPort: LongWord; WaitPin: LongWord;
  WaitState: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_MasterSPI_TransferArrayWS2(ByVal DeviceIdx As UInteger,
  ByRef WriteArray As Byte, ByRef ReadArray As Byte, ByVal Length As UInteger,
  ByVal ChipSelectPort As UInteger, ByVal ChipSelectBit As UInteger,
  ByVal WaitPort As UInteger, ByVal WaitPin As UInteger,
  ByVal WaitState As UInteger) As Integer

Function eProDas_MasterSPI_TransferArrayOUT_WS2(ByVal DeviceIdx As UInteger,
  ByRef WriteArray As Byte, ByVal Length As UInteger,
  ByVal ChipSelectPort As UInteger, ByVal ChipSelectBit As UInteger,
  ByVal WaitPort As UInteger, ByVal WaitPin As UInteger,
  ByVal WaitState As UInteger) As Integer

Function eProDas_MasterSPI_TransferArrayIN_WS2(ByVal DeviceIdx As UInteger,
  ByRef ReadArray As Byte, ByVal Length As UInteger,
  ByVal ChipSelectPort As UInteger, ByVal ChipSelectBit As UInteger,
  ByVal WaitPort As UInteger, ByVal WaitPin As UInteger,
  ByVal WaitState As UInteger) As Integer
```

...please, examine the difference between the `TransferByteWS` and `TransferByteWS2` group of functions to grasp the idea.

### 16.5.9 StatusSPI

Prototype in C/C++

```
int eProDas_StatusSPI(unsigned int DeviceIdx, unsigned int &BufferFull,
  unsigned int &WriteCollision, unsigned int &ReceiveOverflow);
```

Prototype in Delphi

```
function eProDas_StatusSPI(DeviceIdx: LongWord; var BufferFull: LongWord;
  var WriteCollision: LongWord; var ReceiveOverflow: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_StatusSPI(ByVal DeviceIdx As UInteger,
  ByRef BufferFull As UInteger, ByRef WriteCollision As UInteger,
  ByRef ReceiveOverflow As UInteger) As Integer
```

If you use eProDas implemented functions for transferring data over SPI bus, you actually do not need the `StatusSPI` function, since eProDas stack handles all necessary SPI status states/transitions internally. Still, if you are the hacker with acutely red eyes you probably cannot stick with offered solutions and you have to do something by yourself. In this case you need the access to the internal state of the PIC's SPI module, which this function returns through the three parameters.

When each byte is transferred over SPI bus the `BufferFull` parameter holds a non-zero value till you consume the byte that the addressed periphery sent to the PIC. All eProDas transfer functions automatically read the received value in order to deliver it to you, so they never leave SPI module in `BufferFull` state.

The parameter `WriteCollision` holds a non-zero value if you intend to send another byte over SPI before the currently active transfer completed. Again, all eProDas functions wait till the end of SPI cycle so they do not even give you the chance to try something so sinful.

The parameter `ReceiveOverflow` holds a non-zero value if the new data arrived from the periphery during the `BufferFull` state is present. This cannot happen with any eProDas SPI transfer function where PIC is in SPI master mode, however it can happen if PIC is the slave and the other master sends data too fast for your application to consume.

## 16.6 Using SPI and USART modules together

Pin RC7 is shared by both SPI (section 15.6) and USART (section 15.7) modules. In the former case the pin is serial output RC7/SDO (therefore configured as digital output), whereas in the later case it is serial input RC7/RX (therefore configured as digital input). Such selection is truly misfortunate and according to some humanitarian standards it could be annotated as PIC's bug.

If you intend to use both modules in your application, you must take absolute care that your external USART periphery does not exercise any excitation of pin RC7 while SPI module is enabled. In some cases even certain hardware solutions may be necessary, such as TRI state buffers. Note also that SPI module must not excite the unfortunate pin RC7 while USART module is receiving some data from the periphery. eProDas functions help you a bit with this stupid shared-pin issue by putting certain constraints on your application about when both modules in question can be enabled. The story now unfolds.

Functions for configuring SPI module (sections from 15.6.1 to 15.6.3) possess parameter `EnableOutput`, which gives you the opportunity of disabling SPI output, by means of which SPI module entirely relinquishes the control of pin RC7 to USART module and the clash is prevented entirely. Of course, you can apply such solution only if you do not need to send any data to SPI periphery, like in the case of external AD chip, where you only collect AD results but you do not need to configure the chip, for example to select different multiplexed channel.

We can approach the same solution from the other side of the table. Functions for configuring USART (section 15.7.1.1) also possess parameter `EnableInput`, by which you can disable reception of data on USART input pin, by means of which pin RC7 comes under total control of SPI module. Needless to say, this is possible only if you intend to send some data to USART hooked periphery but you do not need to receive anything from these devices.

### 16.6.1 Okay, the above solutions are not suitable for my case. What now?

Boy, you are a demanding guy. Now things are becoming weird. eProDas does not let you fire up the SPI module (by calling any of the functions in sections from 15.6.1 to 15.6.3) with enabled option `EnableOutput`, if USART module is **already** switched on and it's respective option `EnableInput` is active. Nonetheless, you **must** call the function for SPI configuration of your selection (with non-zero value of `EnableOutput`), since the module is configured correctly in the process although it is being held in a switched off state.

Similarly, is SPI is **already** enabled with option `EnableOutput` then eProDas does not let you enable USART module with option `EnableInput`. Nonetheless, the module is correctly configured by the function for configuring it (section 15.7.1.1), although it is held in an inactive state and for that matter you **must** call the function for USART configuration.

When you want to enable the disabled (but otherwise correctly configured) periphery, use the appropriate of the functions `SwitchFrom_USART_To_SPI` and `SwitchFrom_SPI_To_USART`. The former first disables USART module, then it configures pin RC7 as digital output and finally enables SPI module. The latter first disables SPI module, configures pin RC7 as digital input and finally enables USART module.

#### 16.6.1.1 SwitchFrom_USART_To_SPI, SwitchFrom_SPI_To_USART

Prototype in C/C++

```
int eProDas_SwitchFrom_USART_To_SPI(unsigned int DeviceIdx);

int eProDas_SwitchFrom_SPI_To_USART(unsigned int DeviceIdx);
```

Prototype in Delphi

```
function eProDas_SwitchFrom_USART_To_SPI(DeviceIdx: LongWord):integer;

function eProDas_SwitchFrom_SPI_To_USART(DeviceIdx: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_SwitchFrom_USART_To_SPI(ByVal DeviceIdx As UInteger) As Integer

Function eProDas_SwitchFrom_SPI_To_USART(ByVal DeviceIdx As UInteger) As Integer
```

**Note 1.** Your external USART periphery must not excite pin RC7 prior the function `SwitchFrom_SPI_To_USART` does its job, since before the switch is over the pin RC7 is configured as digital output and a short circuit condition would occur with potentially disastrous consequences for your hardware and living beings in the surrounding. Similarly, your USART periphery must stop exciting the pin RC7 prior to calling the function `SwitchFrom_USART_To_SPI`.

**Note 2.** Both of the "switching" functions rely on prior correct configuration of modules; therefore you **must not** call **any** of these functions if **any** of the associated modules is not configured and ready to be enabled. If you only intend to use one of the two modules, there are no described pin RC7 clashes and you do not need these functions. Use the module of your selection freely and entirely ignore the issue that this section is babbling about.

### 16.6.1.2 SwitchFrom_USART_To_SPI_TogglePin*XXX*

Prototype in C/C++

```
int eProDas_SwitchFrom_USART_To_SPI_TogglePinBefore(unsigned int DeviceIdx,
  unsigned int Port, unsigned int Pin);

int eProDas_SwitchFrom_USART_To_SPI_TogglePinAfter(unsigned int DeviceIdx,
  unsigned int Port, unsigned int Pin);

int eProDas_SwitchFrom_USART_To_SPI_TogglePinBeforeAfter(unsigned int DeviceIdx,
  unsigned int PortBefore, unsigned int PinBefore, unsigned int PortAfter,
  unsigned int PinAfter);
```

Prototype in Delphi

```
function eProDas_SwitchFrom_USART_To_SPI_TogglePinBefore(DeviceIdx: LongWord;
  Port: LongWord; Pin: LongWord):integer;

function eProDas_SwitchFrom_USART_To_SPI_TogglePinAfter(DeviceIdx: LongWord;
  Port: LongWord; Pin: LongWord):integer;

function eProDas_SwitchFrom_USART_To_SPI_TogglePinBeforeAfter(DeviceIdx: LongWord;
  PortBefore: LongWord; PinBefore: LongWord; PortAfter: LongWord;
  PinAfter: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_SwitchFrom_USART_To_SPI_TogglePinBefore
  (ByVal DeviceIdx As UInteger, ByVal Port As UInteger, ByVal Pin As UInteger)
  As Integer

Function eProDas_SwitchFrom_USART_To_SPI_TogglePinAfter
  (ByVal DeviceIdx As UInteger, ByVal Port As UInteger, ByVal Pin As UInteger)
  As Integer

Function eProDas_SwitchFrom_USART_To_SPI_TogglePinBeforeAfter
  (ByVal DeviceIdx As UInteger, ByVal PortBefore As UInteger,
  ByVal PinBefore As UInteger, ByVal PortAfter As UInteger,
  ByVal PinAfter As UInteger) As Integer
```

Sometimes you must resort to hardware mechanisms for preventing collisions between SPI aware receiver and USART aware transmitter. An example of such solution is a TRI state buffer through which e.g. USART signal that drives PIC's pin RC7 is passed or not. The feed through of such external buffer is often controlled by certain state on a certain pin. Upon switching the operation between SPI and USART modules the state of this very pin must be altered to properly open or close the signal's door and the functions in this section are here to help you with this task.

Both of the functions with the name pattern SwitchFrom_*XXX*_To_*YYY*_TogglePin*Before* toggle the state of a selected pin before the switch is made. Similarly, the function name pattern SwitchFrom_*XXX*_To_*YYY*_TogglePin*After*, reveals that the pin toggling is done after the switch is over. Finally, functions with name pattern SwitchFrom_*XXX*_To_*YYY*_TogglePin*BeforeAfter*, toggle the state of some pin before the switch is made and they also toggle (the same or some other) pin after the switch is over. The pin in question is specified in a usual way with the combination of port and pin/bit index.

### 16.6.1.3 SwitchFrom_SPI_To_USART_TogglePin*XXX*

Prototype in C/C++

```
int eProDas_SwitchFrom_SPI_To_USART_TogglePinBefore(unsigned int DeviceIdx,
  unsigned int Port, unsigned int Pin);

int eProDas_SwitchFrom_SPI_To_USART_TogglePinAfter(unsigned int DeviceIdx,
  unsigned int Port, unsigned int Pin);

int eProDas_SwitchFrom_SPI_To_USART_TogglePinBeforeAfter(unsigned int DeviceIdx,
  unsigned int PortBefore, unsigned int PinBefore, unsigned int PortAfter,
  unsigned int PinAfter);
```

Prototype in Delphi

```
function eProDas_SwitchFrom_SPI_To_USART_TogglePinBefore(DeviceIdx: LongWord;
  Port: LongWord; Pin: LongWord):integer;

function eProDas_SwitchFrom_SPI_To_USART_TogglePinAfter(DeviceIdx: LongWord;
  Port: LongWord; Pin: LongWord):integer;

function eProDas_SwitchFrom_SPI_To_USART_TogglePinBeforeAfter(DeviceIdx: LongWord;
  PortBefore: LongWord; PinBefore: LongWord; PortAfter: LongWord;
  PinAfter: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_SwitchFrom_SPI_To_USART_TogglePinBefore
  (ByVal DeviceIdx As UInteger, ByVal Port As UInteger, ByVal Pin As UInteger)
  As Integer

Function eProDas_SwitchFrom_SPI_To_USART_TogglePinAfter
  (ByVal DeviceIdx As UInteger, ByVal Port As UInteger, ByVal Pin As UInteger)
  As Integer

Function eProDas_SwitchFrom_SPI_To_USART_TogglePinBeforeAfter
  (ByVal DeviceIdx As UInteger, ByVal PortBefore As UInteger,
  ByVal PinBefore As UInteger, ByVal PortAfter As UInteger,
  ByVal PinAfter As UInteger) As Integer
```

## 16.7 Measurement of delay

There exists certain group of experiments, where you excite (toggle) some digital output pin and measure the amount of time that passes by until the external periphery responds by toggling the state of some other digital input pin. An example is ultrasound distance measurement (section 17.2.1).

With eProDas you can easily achieve the described functionality, since the API directly supports it. Let us start with the simple albeit not the most flexible way of doing the task.

### 16.7.1 MeasureDelay_HighRes

Prototype in C/C++

```
int eProDas_MeasureDelay_HighRes(unsigned int DeviceIdx,
  unsigned int ExcitationPort, unsigned int ExcitationPin, unsigned int ConfigPins,
  unsigned int ActiveExcitationState, unsigned int ActiveDetectedState,
  unsigned int WaitForInactiveState, unsigned int StopExcitingAtStart,
  unsigned int TimeOut, unsigned __int64 &Delay_12MHz_Ticks);
```

Prototype in Delphi

```
function eProDas_MeasureDelay_HighRes(DeviceIdx: LongWord;
  ExcitationPort: LongWord; ExcitationPin: LongWord; ConfigPins: LongWord;
  ActiveExcitationState: LongWord; ActiveDetectedState: LongWord;
  WaitForInactiveState: LongWord; StopExcitingAtStart: LongWord;
  TimeOut: LongWord; var Delay_12MHz_Ticks: Int64):integer;
```

Prototype in VisualBasic

```
Function eProDas_MeasureDelay_HighRes(ByVal DeviceIdx As UInteger,
  ByVal ExcitationPort As UInteger, ByVal ExcitationPin As UInteger,
  ByVal ConfigPins As UInteger, ByVal ActiveExcitationState As UInteger,
  ByVal ActiveDetectedState As UInteger, ByVal WaitForInactiveState As UInteger,
  ByVal StopExcitingAtStart As UInteger, ByVal TimeOut As UInteger,
  ByRef Delay_12MHz_Ticks As Int64) As Integer
```

The literal `HighRes` in the function name denotes measurement of delay with as high resolution as PIC's hardware permits it. The input signal is sampled at a frequency of 12 MHz, so there is only about 83.3 ns of uncertainty of the exact response occurrence.

**Note.** The shortest measurable delay is 1 µs (12 ticks of 12 MHZ clock), although the resolution (uncertainty) of the result is 83.3 ns.

The drawback of high-resolution delay measurement is that certain PIC's hardware capabilities need to be exploited for the task and for that matter module PWM 2 (section 15.5.4) cannot be activated during the working hours of this function. Also, the input pin, which is sampled, is not an arbitrary one but you must resort to the same pin that is used as the output of PWM 2 module. The pin can be either RC1 or RB3, which can be configured in a non-volatile way as the section 15.5.7 describes; see also the section 24.1.1 near the end.

**Note.** eProDas does not check whether the module PWM2 is active at the time of calling this function, since this would slow things down a bit (one more USB packet into each direction). It is solely your responsibility to assure that module PWM2 is disabled prior to calling this function or eProDas device will start behaving in an erratic and unpredictable way.

With parameters `ExcitationPort` and `ExcitationPin` you specify the output pin that eProDas excites at the start of the measurement. The related parameter `ActiveExcitationState` reveals the state (high if the parameter is non-zero) that eProDas sets at the instant of the start.

With a non-zero value of the parameter `ConfigPins` you instruct the device to automatically configure the input pin (either RC1 or RB3, whichever is configured for the task) as digital input and the selected excitation pin as digital output. eProDas also assures that the state of the excitation pin is held in a non-active state (the opposite state of the one that is specified by `ActiveExcitationState`) in order not to prematurely excite the periphery.

**Note.** Pins' configuration takes some time (a couple of USB packets into each direction, since configuration of input pin needs to be checked first) and for that matter we recommend you to use this option at most upon the first execution of the measurement in a row. A simple Boolean-semantic indicator, which the application clears after the first measurement, may be suitable to prevent further idempotent reconfigurations of the pins.

The parameter `ActiveDetectedState` specifies the state on the input pin that eProDas waits for. Again, a non-zero value specifies high logic state.

With a non-zero value of the parameter `WaitForInactiveState`, you instruct eProDas not to start measurement until the input pin (either RC1 or RB3) is in inactive logic state (the opposite of the state that is specified with the parameter `ActiveDetectedState`).

**Note.** The waiting for the inactive state takes arbitrarily long. Therefore, if you e.g. hardwire the input pin to the appropriate power supply rail, your device would be frozen until you cut the wire. The soon to be described parameter `TimeOut` does not cover this initial waiting either.

With a non-zero value of the parameter `StopExcitingAtStart` eProDas stops the excitation on the output pin shortly after the measurement starts (about 500 ns after the start), otherwise the excitation stops after the appropriate state on the input pin is detected and the measurement is over.

With the parameter `TimeOut` you specify the amount of time (in milliseconds) that you are willing to wait for the response from the external periphery. Note that eProDas will wait **at least** the specified amount of time for the response but it may wait a couple of milliseconds longer.

The measured delay is returned through the parameter `Delay_12MHz_Ticks`, which holds the number of 12 MHz clock periods that the delay spanned through; if the operation timed out, the number zero is returned instead. Please, read comments about PIC's clock accuracy in the section 19.7.

Therefore, to calculate the delay in seconds, divide the returned number with 12,000,000. Similarly, to calculate the delay in milliseconds, divide the returned number with 12,000, etc. Of course, at least with short delays a floating-point arithmetic is obligated or you will always get the delay of zero.

**Note.** Although the amount of delay is returned as 64-bit integer, internally eProDas uses 40-bit counter for counting 12 MHz ticks. Therefore, after slightly more than one day or more accurately $2^{40}/\left(12\cdot 10^{6}\right)$ seconds, the counter rolls over without any warnings. If you need to measure such long delays, you need to combine the described measurement with electronic calendar.

#### 16.7.1.1 Two exemplar situations

The myriad of tedious-to-remember parameters tries to cover a broad spectrum of situations, where measurements of delay are needed. One such situation is ultrasound distance measurement (section 17.2.1), where eProDas excites some output pin by means of which an attached ultrasound module starts generating sound waves. Then eProDas measures the delay that is needed for ultrasound sensor to detect its own echo. By knowing the speed of sound propagation we can then calculate the distance of a near by object, a wall, etc.

In such cases you have to configure the measurement in the following way. The excitation must last until the sound echo is detected (zero value of the parameter `StopExcitingAtStart`), unless the users' manual for your module instructs it otherwise. Also, before the measurement starts you can wait until the previous echo is over (non-zero value of the parameter `WaitForInactiveState`).

Another typical situation is measurement of the length of the pulse that is produced by a monostable multivibrator; say a well-known and time-honoured timer-for-the-masses 555 in a proper regime of operation. With timer 555 you cannot measure the delay with the same settings as they were described in the previous paragraph.

Namely, 555 switches its output to the inactive state (in this terminology the active state is the one that stops the measurement) only after it is triggered (excited in this terminology). Therefore, you must not wait for the inactive state to occur (zero value of the parameter `WaitForInactiveState`) or you would wait forever. In addition, the excitation must stop after 555 is triggered (non-zero value of the parameter `StopExcitingAtStart`), otherwise the length of the produced pulse would extend into the next ice age where the electrical power ceases to supply your circuit.

### 16.7.2 StartMeasureDelay_HighRes

Prototype in C/C++

```
int eProDas_StartMeasureDelay_HighRes(unsigned int DeviceIdx,
  unsigned int ExcitationPort, unsigned int ExcitationPin, unsigned int ConfigPins,
  unsigned int ActiveExcitationState, unsigned int ActiveDetectedState,
  unsigned int WaitForInactiveState, unsigned int StopExcitingAtStart);
```

Prototype in Delphi

```
function eProDas_StartMeasureDelay_HighRes(DeviceIdx: LongWord;
  ExcitationPort: LongWord; ExcitationPin: LongWord; ConfigPins: LongWord;
  ActiveExcitationState: LongWord; ActiveDetectedState: LongWord;
  WaitForInactiveState: LongWord; StopExcitingAtStart: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_StartMeasureDelay_HighRes(ByVal DeviceIdx As UInteger,
  ByVal ExcitationPort As UInteger, ByVal ExcitationPin As UInteger,
  ByVal ConfigPins As UInteger, ByVal ActiveExcitationState As UInteger,
  ByVal ActiveDetectedState As UInteger, ByVal WaitForInactiveState As UInteger,
  ByVal StopExcitingAtStart As UInteger) As Integer
```

When the expected delays are a rather long ones (on the orders of seconds, minutes or maybe hours) you probably want to do something else with your application while it is waiting for the response (however, you **cannot** do anything else with your eProDas device). For that matter you can split the functionality of the previous function into several parts so that the application is not frozen while the measurement is in progress.

With the function `StartMeasureDelay_HighRes` you only start the measurement, but you gain the control back immediately instead of after the response on the input pin occurs. The meaning of the parameters is the same as with the function `MeasureDelay_HighRes`.

**Note.** As it was described in the previous section, before the measurement starts eProDas waits for the inactive state on the input pin, when the value of the parameter `WaitForInactiveState` is non-zero. Until this state is detected, the function does not return, since only then the measurement starts.

### 16.7.3 MonitorMeasureDelay_HighRes

Prototype in C/C++

```
int eProDas_MonitorMeasureDelay_HighRes(unsigned int DeviceIdx,
  unsigned int &MeasurementInProgress, unsigned int &MeasurementFinished,
  unsigned __int64 &Delay_12MHz_Ticks);
```

Prototype in Delphi

```
function eProDas_MonitorMeasureDelay_HighRes(DeviceIdx: LongWord;
  var MeasurementInProgress: LongWord; var MeasurementFinished: LongWord;
  var Delay_12MHz_Ticks: Int64):integer;
```

Prototype in VisualBasic

```
Function eProDas_MonitorMeasureDelay_HighRes(ByVal DeviceIdx As UInteger,
  ByRef MeasurementInProgress As UInteger, ByRef MeasurementFinished As UInteger,
  ByRef Delay_12MHz_Ticks As Int64) As Integer
```

With this function you monitor the progress of the previously started measurement. The current state of affairs is returned through the three listed parameters. The parameter `MeasurementInProgress` is set to a non-zero value if the measurement is currently going on.

A non-zero value of the parameter `MeasurementFinished` indicates that the measurement has finished and that the delay is reported to you through the parameter `Delay_12MHz_Ticks`.

If you call this function without previously starting the measurement with the function `StartMeasureDelay_HighRes` or if you previously terminated the measurement with the next to be described function, then both of the parameters `MeasurementInProgress` and `MeasurementFinished` hold zero value.

**Note.** The accuracy of measurement does not depend on the frequency with which you monitor the measurement. eProDas monitors the input pin at an unchanged pace even if you do not check the progress for years (in which case you should pay attention to your delays, too ☺).

### 16.7.4 StopMeasureDelay_HighRes

Prototype in C/C++

```
int eProDas_StopMeasureDelay_HighRes(unsigned int DeviceIdx);
```

Prototype in Delphi

```
function eProDas_StopMeasureDelay_HighRes(DeviceIdx: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_StopMeasureDelay_HighRes(ByVal DeviceIdx As UInteger) As Integer
```

If your patience is exhausted, you can prematurely stop the delay measurement with this function.

**Note.** If the measurement completes successfully, eProDas automatically stops the measurement activities and for that matter you do not have to (although you can) call this function.

### 16.7.5 MeasureDelay_MidRes

Prototype in C/C++

```
int eProDas_MeasureDelay_MidRes(unsigned int DeviceIdx,
  unsigned int ExcitationPort, unsigned int ExcitationPin,
  unsigned int MonitoringPort, unsigned int MonitoringPin, unsigned int ConfigPins,
  unsigned int ActiveExcitationState, unsigned int ActiveDetectedState,
  unsigned int WaitForInactiveState, unsigned int StopExcitingAtStart,
  unsigned int &Delay_12MHz_Ticks);
```

Prototype in Delphi

```
function eProDas_MeasureDelay_MidRes(DeviceIdx: LongWord; ExcitationPort: LongWord;
  ExcitationPin: LongWord; MonitoringPort: LongWord; MonitoringPin: LongWord;
  ConfigPins: LongWord; ActiveExcitationState: LongWord;
  ActiveDetectedState: LongWord;  WaitForInactiveState: LongWord;
  StopExcitingAtStart: LongWord; var Delay_12MHz_Ticks: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_MeasureDelay_MidRes(ByVal DeviceIdx As UInteger,
  ByVal ExcitationPort As UInteger, ByVal ExcitationPin As UInteger,
  ByVal MonitoringPort As UInteger, ByVal MonitoringPin As UInteger,
  ByVal ConfigPins As UInteger, ByVal ActiveExcitationState As UInteger,
  ByVal ActiveDetectedState As UInteger, ByVal WaitForInactiveState As UInteger,
  ByVal StopExcitingAtStart As UInteger, ByRef Delay_12MHz_Ticks As UInteger)
  As Integer
```

When the choice of input pin RC1 or RB3 is too restrictive for your case, you need to resort to delay measurement by utilizing purely software techniques. This function does not use any PIC's (signal transition) capture capabilities for detecting the change of the input pin state, but it instead inspects the state in a tight program loop. Consequently, the uncertainty of delay is degraded from 83.3 ns to 416.7 ns, which is five times worse than in the case of high-resolution delay measurement.
**Note.** The shortest measurable delay is 1.75 µs (21 ticks of 12 MHZ clock).

The pin to be monitored is an arbitrary one, and for that matter two new parameters `MonitoringPort` and `MonitoringPin` are introduced in comparison to the high resolution delay measurement configuration (section 16.7.1). Also, the `TimeOut` parameter is missing since the device does not have the time to check for timeout without further degrading the measurement uncertainty. Other parameters have the same meaning as with the high-resolution delay measurement.

The result of measurement is returned as the number of 12 MHz periods. However, this time only 32-bit counter is used for counting the ticks. Therefore, the maximal permissible delay is about 350 seconds after which the rollover happens without any warnings and indications.

### 16.7.6 MeasureDelay_LowRes

Prototype in C/C++

```
int eProDas_MeasureDelay_LowRes(unsigned int DeviceIdx,
  unsigned int ExcitationPort, unsigned int ExcitationPin,
  unsigned int MonitoringPort, unsigned int MonitoringPin, unsigned int ConfigPins,
  unsigned int ActiveExcitationState, unsigned int ActiveDetectedState,
  unsigned int WaitForInactiveState, unsigned int StopExcitingAtStart,
  unsigned int TimeOut, unsigned __int64 &Delay_12MHz_Ticks);
```

Prototype in Delphi

```
function eProDas_MeasureDelay_LowRes(DeviceIdx: LongWord;
  ExcitationPort: LongWord; ExcitationPin: LongWord; MonitoringPort: LongWord;
  MonitoringPin: LongWord; ConfigPins: LongWord; ActiveExcitationState: LongWord;
  ActiveDetectedState: LongWord;  WaitForInactiveState: LongWord;
  StopExcitingAtStart: LongWord; TimeOut: LongWord;
  var Delay_12MHz_Ticks: Int64):integer;
```

Prototype in VisualBasic

```
Function eProDas_MeasureDelay_LowRes(Byval DeviceIdx As UInteger,
  ByVal ExcitationPort As UInteger, ByVal ExcitationPin As UInteger,
  ByVal MonitoringPort As UInteger, ByVal MonitoringPin As UInteger,
  ByVal ConfigPins As UInteger, ByVal ActiveExcitationState As UInteger,
  ByVal ActiveDetectedState As UInteger, ByVal WaitForInactiveState As UInteger,
  ByVal StopExcitingAtStart As UInteger, ByVal TimeOut As UInteger,
  ByRef Delay_12MHz_Ticks As Int64) As Integer
```

The literal `LowRes` in the function name denotes measurement of delay with a low resolution. The uncertainty of the result is 64 periods of 12 MHz clock or 5,333 ns. The arbitrary input pin is sampled purely in software and the sampling process permits monitoring of the activity as well as a whole-day long measurements.

**Note.** The shortest measurable delay is 2,25 µs (27 ticks of 12 MHZ clock).

The parameters have the same meaning as in the case of the function `MeasureDelay_HighRes` (section 16.7.1), except the parameters `MonitoringPort` and `MonitoringPin`, which have the same meaning as in the case of the function `MeasureDelay_MidRes` (section 16.7.5).

### 16.7.7 StartMeasureDelay_LowRes

Prototype in C/C++

```
int eProDas_StartMeasureDelay_LowRes(unsigned int DeviceIdx,
  unsigned int ExcitationPort, unsigned int ExcitationPin,
  unsigned int MonitoringPort, unsigned int MonitoringPin, unsigned int ConfigPins,
  unsigned int ActiveExcitationState, unsigned int ActiveDetectedState,
  unsigned int WaitForInactiveState, unsigned int StopExcitingAtStart);
```

Prototype in Delphi

```
function eProDas_StartMeasureDelay_LowRes(DeviceIdx: LongWord;
  ExcitationPort: LongWord; ExcitationPin: LongWord; MonitoringPort: LongWord;
  MonitoringPin: LongWord; ConfigPins: LongWord; ActiveExcitationState: LongWord;
  ActiveDetectedState: LongWord;  WaitForInactiveState: LongWord;
  StopExcitingAtStart: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_StartMeasureDelay_LowRes(ByVal DeviceIdx As UInteger,
  ByVal ExcitationPort As UInteger, ByVal ExcitationPin As UInteger,
  ByVal MonitoringPort As UInteger, ByVal MonitoringPin As UInteger,
  ByVal ConfigPins As UInteger, ByVal ActiveExcitationState As UInteger,
  ByVal ActiveDetectedState As UInteger, ByVal WaitForInactiveState As UInteger,
  ByVal StopExcitingAtStart As UInteger) As Integer
```

The function parallels the one with the name `StartMeasureDelay_HighRes` (section 16.7.2), except that it starts a low resolution delay measurement. The meaning of the parameters should be obvious from the previous subsections.

### 16.7.8 MonitorMeasureDelay_LowRes

Prototype in C/C++

```
int eProDas_MonitorMeasureDelay_LowRes(unsigned int DeviceIdx,
  unsigned int &MeasurementInProgress, unsigned int &MeasurementFinished,
  unsigned __int64 &Delay_12MHz_Ticks);
```

Prototype in Delphi

```
function eProDas_MonitorMeasureDelay_LowRes(DeviceIdx: LongWord;
  var MeasurementInProgress: LongWord; var MeasurementFinished: LongWord;
  var Delay_12MHz_Ticks: Int64):integer;
```

Prototype in VisualBasic

```
Function eProDas_MonitorMeasureDelay_LowRes(ByVal DeviceIdx As UInteger,
  ByRef MeasurementInProgress As UInteger, ByRef MeasurementFinished As UInteger,
  ByRef Delay_12MHz_Ticks As Int64) As Integer
```

With this function you can monitor the progress of the previously started low resolution delay measurement in a completely analogous way as the function `MonitorMeasureDelay_HighRes` (section 16.7.3) does it for the high resolution delay measurement.

### 16.7.9 StopMeasureDelay_LowRes

Prototype in C/C++

```
int eProDas_StopMeasureDelay_LowRes(unsigned int DeviceIdx);
```

Prototype in Delphi

```
function eProDas_StopMeasureDelay_LowRes(DeviceIdx: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_StopMeasureDelay_LowRes(ByVal DeviceIdx As UInteger) As Integer
```

With this function you can stop the previously started low resolution delay measurement in a completely analogous way as the function `StopMeasureDelay_HighRes` (section 16.7.4) does it for the case of a high resolution delay measurement.

# 17 Built-in support for external periphery

When working with external AD converters, DA converters and other periphery you must do a tedious job of configuring parameters and forming data exchange chunks according to the specifications of the vary periphery. In addition, during actual communication the exchanged bytes must be properly interpreted to extract the meaningful result.

To help you with the task and to make your eProDas experience as pleasant as possible we built the support for some of the external integrated circuits into the `eProDas.DLL`. Of course, it is literally impossible to support every chip that you may come along with, however some of them implement similar or equal communication specifications, any you may use the functions in this sections for the task even if they are not designed specifically for your specific part. Please, compare datasheets of your chips with the supported ones to see what can be done with the implemented functionality.

In addition, please tell us about the chips or modules that you use since we are interested in hearing from you. If our scare time permits us, we will implement support for your propositions in the future releases of the eProDas stack.

**Note.** Every chip has certain performance capabilities (like maximal sample rate of AD converter), which eProDas often cannot approach even remotely close due to many bottlenecks in the system (overhead of interpretation of user’s commands, USB scheduling policy …). We tried hard to give eProDas as much capabilities and speed as feasible for the price, but we cannot do miracles. To squeeze every last possible inch of performance out of your hardware, try to implement the system by using periodic actions of chapter 19.

## 17.1 Support for SPI enabled integrated circuits

Every SPI enabled integrated circuit connects to PIC in a similar way. Always connect chip’s SPI CLK signal to PIC’s RB1, chip’s SPI Din to PIC’s RC7 and chip’s SPI Dout to PIC’s RB0 (often certain chip possesses only Din or Dout in which case simply do not connect the missing pin ☺). In many cases you must also connect *ChipSelect* signal to some address decoding logic (section 16.2) or directly to a free PIC’s pin (in simple cases). Some chips require still additional connections, which are usually again connected to a spare PIC’s pin. Additional connections from more than one chip can connect to the same PIC’s pin since *ChipSelect*s assure that only selected periphery responds to the intended request. The exception is asynchronous pins that some SPI chips posses.

All SPI functions in this chapter can automatically configure or handle *ChipSelect* in the cases where this signal is connected directly to some PIC’s pin. When configuring SPI module (functions like `SetInternalSPI_MasterMode_`*`XXX`*), one of the PIC’s pin can be configured as digital output and its state set to the appropriate logic level (usually 1). The pin in question is specified in a usual way by setting parameters `ChipSelectPort` and `ChipSelectBit` (see section 15.6.2). If you do not want this functionality, set the value of `ChipSelectPort` to 1 000 or more and pin configuration will be skipped.

When communicating with SPI chips, the pin that is connected to *ChipSelect* is toggled to activate the chip prior the communication starts. After the operation on the chip is completed, the same pin is toggled again to deactivate the chip. The `ChipSelectPort` value of 1 000 or more always prevents pin toggling to take place and you can do chip selection by yourself (for example, by utilizing more or less complicated address decoding logic according to section 16.2).

The final topic that we feel the need to mention, are presumably frequent situations whereby you connect many different SPI chips to one poor PIC. How to configure SPI module in this way? Generally, there are two approaches to the problem. You could simply reconfigure the module (section 15.6.6) each time before you initiate the communication with different chip, which is a bit time consuming, unless the need for reconfiguration arises fairly infrequently.

Another approach is to configure SPI module according to the common denominator principle, which basically means that the speed of your SPI module should be configured to the smallest one of all the connected chips. As it turns out this not only leads to smaller data throughput but it may even degrade the performance of some chips (for example, the holding capacitor in AD converter cannot hold the sampled voltage for a long time without noticeable droop, which increases AD conversion error).

**Note.** SPI clock frequency is not the only parameter that needs reconfiguration when switching between different chips. Often at least idle clock state must be reconfigured, too. For example, the soon to be mentioned chip DAC7611 needs high idle state of the clock to work properly, whereas the chip ADS7816 needs a lower one. Therefore, even if you can craft your system is such a way that all chips work with the same SPI clock frequency, your goal of avoiding SPI reconfigurations may not be fully achieved.

The practical situations are diverse and no general purpose suggestions can be done in this place since we have no clue about how and with what external chips you intend to use your eProDas device. Anyway, for the electrical superman as you are, all these challenges are a small potato and the diversity of eProDas API (hopefully) enables you to overcome them all.

### 17.1.1 AD converters MCP3208 and MCP3204 from Microchip

MCP3208(4) is a 100 kSamples/s 12-bit AD converter with 8(4) single ended or 4(2) pseudo-differential multiplexed channels. The following functions exist for your pleasant working with MCP320x without being bothered with too much technicalities.

#### 17.1.1.1 SetInternalSPI_MasterMode_MCP320x

Prototype in C/C++

```
int eProDas_SetInternalSPI_MasterMode_MCP320x(unsigned int DeviceIdx,
  unsigned int ChipSelectPort, unsigned int ChipSelectBit);
```

Prototype in Delphi

```
function eProDas_SetInternalSPI_MasterMode_MCP320x(DeviceIdx: LongWord;
  ChipSelectPort: LongWord; ChipSelectBit: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_SetInternalSPI_MasterMode_MCP320x(ByVal DeviceIdx As UInteger,
  ByVal ChipSelectPort As UInteger, ByVal ChipSelectBit As UInteger) As Integer
```

This function configures PIC’s SPI module according to MCP320x’s specifications. Specifically, it calls the function `SetInternalSPI_MasterMode_CS` with the following parameters: `IN_SampleAtTheEnd` = 0, `ClockIdleStateHigh` = 0, `ClockSpeed` = 3 (controlled by Timer2), `ClockPeriod` = 2, `ClockPrescaler` = 1, `EnableInput` = 1, `EnableOutput` = 1. As you can see, according to this configuration Timer2 period expires on each (`ClockPeriod`+1)*`ClockPrescaler` = 3 period of 12 MHz clock, which gives us the maximal 2 MHz SPI clock that MCP320x can work with (remember, SPI clock is Timer2 clock divided by 2).

#### 17.1.1.2 ReadExternalAD_MCP320x

Prototype in C/C++

```
int eProDas_ReadExternalAD_MCP320x(unsigned int DeviceIdx, unsigned int Channel,
  unsigned int SingleEnded, unsigned int &Result, unsigned int ChipSelectPort,
  unsigned int ChipSelectBit);
```

Prototype in Delphi

```
function eProDas_ReadExternalAD_MCP320x(DeviceIdx: LongWord; Channel: LongWord;
  SingleEnded: LongWord; var Result: LongWord; ChipSelectPort: LongWord;
  ChipSelectBit: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_ReadExternalAD_MCP320x(ByVal DeviceIdx As UInteger,
  ByVal Channel As UInteger, ByVal SingleEnded As UInteger,
  ByRef Result As UInteger, ByVal ChipSelectPort As UInteger,
  ByVal ChipSelectBit As UInteger)As Integer
```

With this function you perform the AD conversion. Parameter `Channel` (from 0 to 7) selects the desired MCP320x's multiplexed channel to be read. The non-zero value of the parameter `SingleEnded` instructs a single-ended AD conversion (i.e. the reading of the voltage difference between the pin and AGND); in the opposite case the two adjacent analog inputs are combined together into the pseudo-differential analog input pair and the voltage difference between them is measured. In the differential regime of operation `Channel` 0 or 1 selects the first input pair, 2 or 3 selects the second one, etc. The result of AD conversion (between 0 and 4095) is returned in the parameter `Result`.

The C/C++ example program (without **demanded** error checking) is as follows. It is assumed that you want to read channel 5 of MCP MCP320x in a single-ended mode and that *ChipSelect* of MCP320x is wired to PIC's pin RB3. Do not forget to open and close device handle (section 14.3), though.

```
  const DeviceIdx = 0;
  unsigned int AD_Result;

  //Configure SPI module for MCP320x and pin RB3 as digital output, set to logic 1
  eProDas_SetInternalSPI_MasterMode_MCP320x(DeviceIdx, eProDas_Index_PORTB, 3);

  //perform AD conversion
  const Channel = 5;
  const SingleEnded = 1;
  eProDas_ReadExternalAD_MCP320x(DeviceIdx, Channel, SingleEnded, AD_Result,
      eProDas_Index_PORTB, 3);

  //voila, consume the AD_Result
```

**Figure 46: Example code fragment for performing AD conversion with the chip MCP3208.**

### 17.1.1.3 ReadExternalAD_MCP320x_Selected

Prototype in C/C++

```
int eProDas_ReadExternalAD_MCP320x_Selected(unsigned int DeviceIdx,
  unsigned int SingleEndedFlags, unsigned int DifferentialFlags,
  unsigned int *Results, unsigned int ChipSelectPort, unsigned int ChipSelectBit);
```

Prototype in Delphi

```
function eProDas_ReadExternalAD_MCP320x_Selected(DeviceIdx: LongWord;
  SingleEndedFlags: LongWord; DifferentialFlags: LongWord; Results: PLongWord;
  ChipSelectPort: LongWord; ChipSelectBit: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_ReadExternalAD_MCP320x_Selected(ByVal DeviceIdx As UInteger,
  ByVal SingleEndedFlags As UInteger, ByVal DifferentialFlags As UInteger,
  ByRef Results As UInteger, ByVal ChipSelectPort As UInteger,
  ByVal ChipSelectBit As UInteger) As Integer
```

Often you need to read several AD channels at once, which could be done by calling the function `ReadExternalAD_MCP320x` several times. However, in this scenario an isolated USB packet travels from host PC to eProDas device and back for each AD reading, which is rather inefficient. A substantially faster execution can be achieved by packing several requests into one USB packet, which the function `ReadExternalAD_MCP320x_Selected` does for you.

The parameter `SingleEndedFlags` is a bit field mask by which you specify the AD inputs (by setting the respective bits from 0 to 7) that need to be sampled in single ended mode. Similarly, the parameter `DifferentialFlags` carry your requests about AD input pairs (bits from 0 to 3) that need to be sampled in differential mode. The specification of channels is completely arbitrary; the only limitation is that at most ten conversions in total can be specified without exhausting the capacity of USB packet buffer.

The results of AD conversions are written to the buffer that is pointed to by the parameter `Results`. The sequence of AD conversions and consequently results in the buffer `Results` is: first, all specified single ended conversions are done from the smallest to the largest AD input number, and then selected differential results follow in the same order.

### 17.1.1.4 ReadExternalAD_MCP320x_All

Prototype in C/C++

```
int eProDas_ReadExternalAD_MCP320x_All(unsigned int DeviceIdx,
  unsigned int SingleEnded, unsigned int *Results, unsigned int ChipSelectPort,
  unsigned int ChipSelectBit);
```

Prototype in Delphi

```
function eProDas_ReadExternalAD_MCP320x_All(DeviceIdx: LongWord;
  SingleEnded: LongWord; Results: PLongWord; ChipSelectPort: LongWord;
  ChipSelectBit: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_ReadExternalAD_MCP320x_All(ByVal DeviceIdx As UInteger,
  ByVal SingleEnded As UInteger, ByRef Results As UInteger,
  ByVal ChipSelectPort As UInteger, ByVal ChipSelectBit As UInteger) As Integer
```

This function is a shorthand for reading all inputs in single ended mode (if parameter `SingleEnded` holds a non-zero value) or all input pairs in differential mode (if parameter `SingleEnded` is zero). The results are delivered in the same way as in the case of the previously described function.

#### 17.1.1.5 Configure_SPI_Clock_MCP320x

Prototype in C/C++

```
int eProDas_Configure_SPI_Clock_MCP320x(unsigned int DeviceIdx);
```

Prototype in Delphi

```
function eProDas_Configure_SPI_Clock_MCP320x(DeviceIdx: LongWord):integer;
```

Prototype in VisualBasic

```
eProDas_Configure_SPI_Clock_MCP320x
```

This function reconfigures SPI clock to the proper settings for working with MCP3208/4. Use it when you have previously exchanged data with different SPI chips and consequently different SPI clock configuration.

### 17.1.2 AD converter ADS7816 from Texas Instruments

ADS7816 is a 200 kSamples/s 12-bit AD converter with one pseudo-differential channel. There are again two functions for utilization of the chip in a straightforward way.

#### 17.1.2.1 SetInternalSPI_MasterMode_ADS7816

Prototype in C/C++

```
int eProDas_SetInternalSPI_MasterMode_ADS7816(unsigned int DeviceIdx,
  unsigned int ChipSelectPort, unsigned int ChipSelectBit);
```

Prototype in Delphi

```
function eProDas_SetInternalSPI_MasterMode_ADS7816(DeviceIdx: LongWord;
  ChipSelectPort: LongWord; ChipSelectBit: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_SetInternalSPI_MasterMode_ADS7816(ByVal DeviceIdx As UInteger,
  ByVal ChipSelectPort As UInteger, ByVal ChipSelectBit As UInteger) As Integer
```

This function configures PIC's SPI module according to ADS7816's specifications. Specifically, it calls the function `SetInternalSPI_MasterMode_CS` with the following parameters: `IN_SampleAtTheEnd` = 0, `ClockIdleStateHigh` = 0, `ClockSpeed` = 1 (3 MHz). The maximal SPI clock that ADS7816 can work with is 3.2 MHz; however PIC is unable to provide exactly this figure.

#### 17.1.2.2 ReadExternalAD_ADS7816

Prototype in C/C++

```
int eProDas_ReadExternalAD_ADS7816(unsigned int DeviceIdx, unsigned int &Result,
  unsigned int ChipSelectPort, unsigned int ChipSelectBit);
```

Prototype in Delphi

```
function eProDas_ReadExternalAD_ADS7816(DeviceIdx: LongWord; var Result: LongWord;
  ChipSelectPort: LongWord; ChipSelectBit: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_ReadExternalAD_ADS7816(ByVal DeviceIdx As UInteger,
  ByRef Result As UInteger, ByVal ChipSelectPort As UInteger,
  ByVal ChipSelectBit As UInteger) As Integer
```

With this function you perform the AD conversion. The result (between 0 and 4095) is returned in the parameter `Result`.

The C/C++ example program (without **demanded** error checking) is as follows. It is assumed that the *ChipSelect* of ADS7816 is wired to PIC's pin RB3. Do not forget to open and close device handle (section 14.3).

```
  const DeviceIdx = 0;
  unsigned int AD_Result;

  //Configure SPI module for ADS7816 and pin RB3 as digital output; set to logic 1
  eProDas_SetInternalSPI_MasterMode_ADS7816(DeviceIdx, eProDas_Index_PORTB, 3);

  //perform AD conversion
  eProDas_ReadExternalAD_ADS8716(DeviceIdx, AD_Result, eProDas_Index_PORTB, 3);

  //voila, consume the AD_Result
```

**Figure 47: Example code fragment for performing AD conversion with the chip ADS7816.**

### 17.1.2.3 Configure_SPI_Clock_ADS7816

Prototype in C/C++

```
int eProDas_Configure_SPI_Clock_ADS7816(unsigned int DeviceIdx);
```

Prototype in Delphi

```
function eProDas_Configure_SPI_Clock_ADS7816(DeviceIdx: LongWord):integer;
```

Prototype in VisualBasic

```
eProDas_Configure_SPI_Clock_ADS7816
```

This function reconfigures SPI clock to the proper settings for working with ADS7816. Use it when you have previously exchanged data with different SPI chips and consequently different SPI clock configuration.

## 17.1.3 DA converter DAC7611 from Texas Instruments

DAC7611 is an interesting 5V single supply 12-bit DA converter with built-in voltage reference and precision output amplifier, which means that it can be used in many situations without external operational amplifier buffer and voltage reference chip. The nominal range of analog output voltage is from 0 V to 4.095 V, therefore one LSB weights 1 mV. Again, you can access the functionality of the chip through two eProDas functions.

### 17.1.3.1 SetInternalSPI_MasterMode_DAC7611

Prototype in C/C++

```
int eProDas_SetInternalSPI_MasterMode_DAC7611(unsigned int DeviceIdx,
  unsigned int ChipSelectPort, unsigned int ChipSelectBit, unsigned int LoadPort,
  unsigned int LoadBit);
```

Prototype in Delphi

```
function eProDas_SetInternalSPI_MasterMode_DAC7611(DeviceIdx: LongWord;
  ChipSelectPort: LongWord; ChipSelectBit: LongWord; LoadPort: LongWord;
  LoadBit: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_SetInternalSPI_MasterMode_DAC7611(ByVal DeviceIdx As UInteger,
  ByVal ChipSelectPort As UInteger, ByVal ChipSelectBit As UInteger,
  ByVal LoadPort As UInteger, ByVal LoadBit As UInteger) As Integer
```

This function configures PIC's SPI module according to DAC7611's specifications. Specifically, it calls the function `SetInternalSPI_MasterMode_CS` with the following parameters: `IN_SampleAtTheEnd` = 0, `ClockIdleStateHigh` = 1, `ClockSpeed` = 0 (12 MHz). The maximal SPI clock that DAC7611 can work with is 20 MHz; however the maximal possible PIC's SPI clock is only 12 MHz.

From the function definition it can be observed that configuration procedure needs two additional parameters `LoadPort` and `LoadPin`. As it turns out DAC7611 requires an additional connection to the PIC to work properly. The complication is due to the double buffering of the DA holding register, which is needed to prevent chaotic behaviour of the analog output voltage during the transfer of the new DA value to the chip. When the transfer is completed, the pulse on a *Load Strobe* pin, which must be connected to the spare PIC's pin, actually changes the analog output voltage. This pin is automatically activated by eProDas upon the completed transfer of DA value.

### 17.1.3.2 WriteExternalDA_DAC7611

Prototype in C/C++

```
int eProDas_WriteExternalDA_DAC7611(unsigned int DeviceIdx, unsigned int Value,
unsigned int ChipSelectPort, unsigned int ChipSelectBit, unsigned int LoadPort,
unsigned int LoadBit);
```

Prototype in Delphi

```
function eProDas_WriteExternalDA_DAC7611(DeviceIdx: LongWord; Value: LongWord;
  ChipSelectPort: LongWord; ChipSelectBit: LongWord; LoadPort: LongWord;
  LoadBit: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_WriteExternalDA_DAC7611(ByVal DeviceIdx As UInteger,
  ByVal Value As UInteger, ByVal ChipSelectPort As UInteger,
  ByVal ChipSelectBit As UInteger, ByVal LoadPort As UInteger,
  ByVal LoadBit As UInteger) As Integer
```

With this function you write new value (between 0 and 4095) to the DA converter. *ChipSelect* as well as *LoadStrobe* are handled automatically unless you specify the value of 1000 or more for *XXX*Port in which case the handling of the respective pin is skipped.

The C/C++ example program (without **demanded** error checking) is as follows. It is assumed that the *ChipSelect* and *LoadStrobe* of DAC7611 are wired to PIC's pins RB3 and RC0, respectively. Say, we want to produce analog voltage of 2.000 V (nominal), in which case we need to write the value of 2 000 to DAC. Do not forget to open and close device handle (section 14.3).

```
  const DeviceIdx = 0;

  //Configure SPI module for DAC7611 + pins RB3 and RC0
  eProDas_SetInternalSPI_MasterMode_DAC7611(DeviceIdx, eProDas_Index_PORTB, 3,
    eProDas_Index_PORTC, 0);

  //write the value of 2000 to DA
  eProDas_WriteExternalDA_DAC7611(DeviceIdx, 2000, eProDas_Index_PORTB, 3,
    eProDas_Index_PORTC, 0);

  //voila, measure the output voltage of DAC7611 with your favorite V-meter
```

**Figure 48: Example code fragment for operating on the chip DAC7611.**

#### 17.1.3.3 Configure_SPI_Clock_DAC7611

Prototype in C/C++
```
int eProDas_Configure_SPI_Clock_DAC7611(unsigned int DeviceIdx);
```

Prototype in Delphi
```
function eProDas_Configure_SPI_Clock_DAC7611(DeviceIdx: LongWord):integer;
```

Prototype in VisualBasic
```
eProDas_Configure_SPI_Clock_DAC7611
```

This function reconfigures SPI clock to the proper settings for working with DAC7611. Use it when you have previously exchanged data with different SPI chips and consequently different SPI clock configuration.

## 17.2 Support for directly digitally connected periphery

This section deals with support for periphery that is directly connected to general-purpose PIC' digital pins, contrary to the periphery that is connected through some sort of official bus (SPI, USART, etc.).

### 17.2.1 Motion detector MD ULI from Vernier

The Vernier's motion detector MD ULI is a purely digitally controlled device. Besides two power-supply wires it has one input pin and one output pin. When the input pin (from the modules' perspective) is excited (high logic state) the module starts to emit ultrasound waves at 40 kHz. When the waves reflect from some nearby object and reach back the receiver, the module excites its output pin (again, high logic state). By measuring the delay between the start of transmission and the start of reception, the distance between the object and the module is measured. Although the procedure seems (oops, sounds) complicated, the complexity is completely hidden from you and encapsulated in the following functions.

**Note.** These functions can be used with any distance sensor, which has the same functionality of the connecting pins. For sensors with negated levels, you have to resort to the more general-purpose functions in the section 16.7.

#### 17.2.1.1 MeasureDistance_Vernier_MD_ULI_HighRes

Prototype in C/C++
```
int eProDas_MeasureDistance_Vernier_MD_ULI_HighRes(unsigned int DeviceIdx,
  unsigned int ExcitationPort, unsigned int ExcitationPin, unsigned int ConfigPins,
  double Speed_MetersPerSecond, double &Distance);
```

Prototype in Delphi
```
function eProDas_MeasureDistance_Vernier_MD_ULI_HighRes(DeviceIdx: LongWord;
  ExcitationPort: LongWord; ExcitationPin: LongWord; ConfigPins: LongWord;
  Speed_MetersPerSecond: Double; var Distance: Double):integer;
```

Prototype in VisualBasic
```
Function eProDas_MeasureDistance_Vernier_MD_ULI_HighRes
  (ByVal DeviceIdx As UInteger, ByVal ExcitationPort As UInteger,
  ByVal ExcitationPin As UInteger, ByVal ConfigPins As UInteger,
  ByVal Speed_MetersPerSecond As Double, ByRef Distance As Double) As Integer
```

This function is only slightly more than a wrapper for the function `MeasureDelay_HighRes` and for that matter we strongly suggest you to read the section 16.7.1 with important explanations regarding the usage. All parameters, except the last two, are passes to the function `MeasureDelay_HighRes` directly and unchanged. In addition, the parameters `ActiveExcitationState`, `ActiveDetectedState` and `TimeOut` of the function `MeasureDelay_HighRes` are set to the values of 1, 1 and 100 (ms), respectively.

Based on the results that the worker-in-the-background function `MeasureDelay_HighRes` returns, the function `MeasureDistance_Vernier_MD_ULI_HighRes` calculates the measured distance according to the specification of the sound wave propagation speed that you specify through the parameter `Speed_MetersPerSecond` (in meters per second, if you wonder ☺).

If the temperature of the air in your laboratory is 25 ºC, set this parameter to 343.4; otherwise adjust the value according to the extremely complicated and boring physics' equations that you can find in the elaborate textbooks about sound waves (even better, ignore the difference and enjoy your life ☺).

#### 17.2.1.2 MeasureDelay_Vernier_MD_ULI_HighRes

Prototype in C/C++

```
int eProDas_MeasureDelay_Vernier_MD_ULI_HighRes(unsigned int DeviceIdx,
  unsigned int ExcitationPort, unsigned int ExcitationPin, unsigned int ConfigPins,
  unsigned __int64 &Delay_12MHz_Ticks);
```

Prototype in Delphi

```
function eProDas_MeasureDelay_Vernier_MD_ULI_HighRes(DeviceIdx: LongWord;
  ExcitationPort: LongWord; ExcitationPin: LongWord; ConfigPins: LongWord;
  var Delay_12MHz_Ticks: Int64):integer;
```

Prototype in VisualBasic

```
Function eProDas_MeasureDelay_Vernier_MD_ULI_HighRes(ByVal DeviceIdx As UInteger,
  ByVal ExcitationPort As UInteger, ByVal ExcitationPin As UInteger,
  ByVal ConfigPins As UInteger, ByRef Delay_12MHz_Ticks As Int64) As Integer
```

This function is really a pure wrapper for the function `MeasureDelay_HighRes` except that it sets some parameters automatically according to the module's specification. Instead of the distance, you get pure information about the propagation delay in 12 MHz ticks.

This function may find usage in specially crafted calculations, where rounding errors of the used floating-point data types may be prohibitively large. Also, the function is more convenient for the experiments, where you do not measure distance but some other property, like the variation of the speed of the air with temperature (not again ☹).

### 17.2.1.3 MeasureDistance_Vernier_MD_ULI_MidRes

Prototype in C/C++

```
int eProDas_MeasureDistance_Vernier_MD_ULI_MidRes(unsigned int DeviceIdx,
  unsigned int ExcitationPort, unsigned int ExcitationPin,
  unsigned int MonitoringPort, unsigned int MonitoringPin, unsigned int ConfigPins,
  double Speed_MetersPerSecond, double &Distance);
```

Prototype in Delphi

```
function eProDas_MeasureDistance_Vernier_MD_ULI_MidRes(DeviceIdx: LongWord;
  ExcitationPort: LongWord; ExcitationPin: LongWord; MonitoringPort: LongWord;
  MonitoringPin: LongWord; ConfigPins: LongWord; Speed_MetersPerSecond: Double;
  var Distance: Double):integer;
```

Prototype in VisualBasic

```
Function eProDas_MeasureDistance_Vernier_MD_ULI_MidRes(ByVal DeviceIdx As UInteger,
  ByVal ExcitationPort As UInteger, ByVal ExcitationPin As UInteger,
  ByVal MonitoringPort As UInteger, ByVal MonitoringPin As UInteger,
  ByVal ConfigPins As UInteger, ByVal Speed_MetersPerSecond As Double,
  ByRef Distance As Double) As Integer
```

The same functionality as in the case of the function `MeasureDistance_Vernier_MD_ULI_HighRes` (section 17.2.1.1) except that this time we utilize the less precise measurement engine of the function `MeasureDelay_MidRes` (section 16.7.5). On the positive side, the input pin is now an arbitrary one (specified by the parameters `MonitoringPort` and `MonitoringPin`).

### 17.2.1.4 MeasureDelay_Vernier_MD_ULI_MidRes

Prototype in C/C++

```
int eProDas_MeasureDelay_Vernier_MD_ULI_MidRes(unsigned int DeviceIdx,
  unsigned int ExcitationPort, unsigned int ExcitationPin,
  unsigned int MonitoringPort, unsigned int MonitoringPin, unsigned int ConfigPins,
  unsigned int &Delay_12MHz_Ticks);
```

Prototype in Delphi

```
function eProDas_MeasureDelay_Vernier_MD_ULI_MidRes(DeviceIdx: LongWord;
  ExcitationPort: LongWord; ExcitationPin: LongWord; MonitoringPort: LongWord;
  MonitoringPin: LongWord; ConfigPins: LongWord;
  var Delay_12MHz_Ticks: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_MeasureDelay_Vernier_MD_ULI_MidRes(ByVal DeviceIdx As UInteger,
  ByVal ExcitationPort As UInteger, ByVal ExcitationPin As UInteger,
  ByVal MonitoringPort As UInteger, ByVal MonitoringPin As UInteger,
  ByVal ConfigPins As UInteger, ByRef Delay_12MHz_Ticks As UInteger) As Integer
```

The same functionality as in the case of the function `MeasureDelay_Vernier_MD_ULI_HighRes` (section 17.2.1.2), except that we utilize the less precise engine of the function `MeasureDelay_MidRes` (section 16.7.5).

# 18 Periodic actions using applications' logic

Arguably, periodic actions are one of the most important concepts of almost any data acquisition, instrumentation or discrete control system. When working with digital or analog oscilloscope we explicitly or implicitly specify sampling rate or frequency of displaying observed signal. Similarly, signal generators and direct digital synthesizers possess settable or predetermined frequency of excitation of their output signal. Whenever at least a part of signal path of a system is implemented digitally the concepts of time domain discretization and value quantization cannot be avoided. The former almost always leads to periodic actions. If eProDas data acquisition system wants to deserve its full name it surely must provide a fair support for this extremely important data acquisition concept.

## 18.1 Capabilities versus user friendliness

Data acquisition systems with one or perhaps a few predetermined scenarios of usage can provide support for periodic actions in a straightforward way. For example, if we deal with a system that consists solely of two AD converters, we hardly imagine much more than options for setting sampling rates of each or both analog inputs and option for inhibiting some of the inputs. The bottom line is that in such cases all options can be specified by a set of variables that a user of the system initializes prior to starting periodic action (in this case sampling of selected input signals).

Contrary to the just described situation, eProDas represents data acquisition systems without predetermined role or likely scenario of usage. The system is deliberately designed to cover as broad set of natural science experiments as possible and therefore the eProDas developers' team cannot have clear clues about how the users of the systems intend to use them.

For example, in some experiments only sampling of AD inputs is needed. Other times one needs to combine this with functionality of voltage comparators and digital I/O ports. Exactly what would someone like to connect to digital ports is further blurred by the rich choice of external digital circuitry that may add functionality to the core system. On one extreme there are pure digital sensors that signal their binary (one bit) or numerical (multi bit) state. On the other extreme there are AD and DA chips with microprocessor interface, other microprocessor subsystems, various communication subsystems (UART, Bluetooth, ...) and many more useful chips and subsystems.

Clearly, if we do not want to confine eProDas system to a small niche of experiments we need to support this entire diversity of periphery. The good news is that a lot has been done when crafting eProDas API and implementing the functionality of the system to enable plugging in a broad range of external circuitry. The bad news is that this broad functionality necessarily makes eProDas periodic actions harder to use. The user is not only requested to set parameters of actions but he or she needs to specify precisely the steps that the system is expected to execute within one period of the actions.

For that matter eProDas system possess special features that were carefully designed to enable the applications developer to periodically execute as broad and unconfined set of actions as feasible (chapter 19). However, for certain non-demanding cases you can go along well with only the so far described functions. This chapter is devoted to giving some hints and tips about this less complicated approach.

## 18.2 Doing periodic actions the ordinary way

In many situations you may implement a desired periodic functionality solely by using functions that are described in previous chapters, i.e. the "ordinary" non-periodic user functions. As an example, let's implement simple periodic sampling of one AD channel with frequency of 1 Hz. A pseudo C++ code for the task looks something like the following programming fragment.

```
//dummy storage for AD results
unsigned int ADResults;

//loop for executing periodic actions
while(Exit_Condition_Is_Not_Fulfilled) {
  //do the action
  eProDas_ReadInternalADChannel(Device_Idx, ADResults);

  //implement clock frequency of periodic action
  Wait_Till_One_Second_Passes();
}
```

**Figure 49: Pseudo code for reading AD channel at a frequency of 1 Hz.**

The main concept here is that periodicity of the action is achieved on a PC side and not on an eProDas side. eProDas device merely executes each request for reading AD channel in isolation without any knowledge of the fact that requests are scheduled by the host PC on a periodic basis.

The second important concept is that the clock for periodic actions is also implemented by a host PC. If the function `Wait_Till_One_Second_Passes()` is changed to `Wait_Till_One_Minute_Passes()` the frequency of periodic action would change from 1 Hz to 1/60 Hz. Regarding this example we conclude that arbitrary periodic actions can already be done by the so far described users' set of functions. This should not be forgotten since in many non-demanding situations this is probably a preferred way to do the job due to a lesser complexity of the applications' code.

### 18.2.1 Limitations of periodic actions on a host PC side

Doing periodic actions on PC side imposes certain limitations and consequently such approach is not capable to cope with more demanding tasks. First, there is a serious USB bandwidth bottleneck which arises from the fact that each USB packet that travels between PC and eProDas device carries only one piece of information at a time (result of AD conversion, state of voltage comparators, state of I/O port...). This results in a significant USB protocol overhead and inefficient scheduling of packets by USB hub. Consequently, you can obtain frequency of at most a couple of hundred hertz of your periodic actions using this approach (section 14.6.1.1, chapter 4).

Second, there is operating system (OS) with its multitasking capabilities in your way. Your eProDas application is only one of the processes that your OS is trying to run in a perceptually simultaneous way. To achieve this functionality your program is constantly pre-empted (interrupted) by OS in order to fairly divide CPU among all processes in the system. Consequently, you cannot predict precisely at which instant of time a certain time-critical part of your periodic actions is going to be executed or interrupted, which leads to unavoidable jitter or small deviations of lengths of isolated periods from the expected (nominal) length. This jitter sums up with the jitter, which results from uncertainty of the exact moment when certain USB packet is scheduled for transmission or reception. Namely, USB protocol demands that packets are enclosed within bus frames (duration of 1 ms in the case of full-speed USB) and since your application (presumably) is not synchronized with USB clock any packet can experience time uncertainty of one frame delay (or more, USB does not guarantee anything).

Despite the stated limitations the concept of periodic actions on a PC side is simple and gives satisfactory results for non-demanding tasks, say for frequencies below 10 Hz (like measurements of static characteristics and observing slowly varying signals).

### 18.2.2 Improper way of doing periodic actions on a host PC side

Let’s go back to the previous pseudo code fragment and make it slightly more real by implementing the pause of one second in a realistic way. The resulting code is as follows.

```
unsigned int ADResults;

while(Exit_Condition_Is_Not_Fulfilled) {
  eProDas_ReadInternalADChannel(Device_Idx, ADResults);

  //implement clock frequency of periodic action
  eProDas_Sleep(1000); //this line is changed          ← ← ← ← ←
}
```

**Figure 50: Reading AD channel at a frequency of 1 Hz using Windows Sleep function.**

As you may or may not know, Windows API implements function `Sleep`, which blocks program execution for the specified amount of time in milliseconds. Therefore, `Sleep(1000)` means that our program (precisely: only the thread that calls `Sleep` function) takes a pause of 1 second, just what we need to implement the desired periodic action at frequency of 1 Hz.

In certain development environments there is no possibility to access the Windows API functions directly and for that matter eProDas implements a wrapper function `eProDas_Sleep` with the following definition. The windows `Sleep` function is called directly and the parameter `MiliSeconds` is passed through completely unchanged.

### 18.2.3 Sleep

Prototype in C/C++

```
void eProDas_Sleep(unsigned int MiliSeconds);
```

Prototype in Delphi

```
procedure eProDas_Sleep(MiliSeconds: LongWord);
```

Prototype in VisualBasic

```
Sub eProDas_Sleep (ByVal MiliSeconds As UInteger)
```

Although the code fragment in the Figure 50 appears simple and correct it in fact demonstrates a rather sloppy and inaccurate way of doing things. Two inherent drawbacks of such design may become a source of dominant timing error. First, sleeping for one second is not exactly what we need. Namely, if we want to execute accurately one periodic action per second, the pause should take into account the duration of function `ReadInternalADChannel`. If, for example, reading of AD result takes 2 ms, the period of periodic action would be 1,000 ms + 2 ms, which gives us frequency of approximately 0,998 Hz (error of 0.2 %). For shorter periods the relative error would increase, but if the period is long enough (on order of minutes, hours or more) the error may not be worth to mention.

The second drawback of the approach exists solely due to the multitasking nature of your operating system. If you ever thoroughly examined description of function `Sleep` you have learned that it guarantees pause of AT LEAST the prescribed amount of time, but NOT the exact amount. This is nothing catastrophic by itself, since any program that is executed under multitasking operating system experiences certain indeterminism of execution timing. The problem is that each time function `Sleep` executes, this small amount of prolonged pause sums up with the previous such deviations and through some long period of time the accumulated error may surprise you.

You may be tempted to remedy the situation by carefully timing the execution during a long period of time, say one hour, to learn about the actual frequency of periodic actions. For example, reading AD channel at frequency of 1 Hz should yield 3,600 samples in one hour. If it turns out that there are only 3,542 samples, you could shorten the sleeping interval accordingly, until the number of samples matches exactly (you hope). Actually, the most you can do is to fine-tune the pause to such value that the average number of samples would be fairly close to your expectations.

Although you can improve the situation by described careful fine-tuning of the application, we strongly recommend you not to waste your precious time on such awkward methods. The first problem is the obvious one: you need an enormous amount of time to measure the performance, adjust the program, measure again, adjust, measure… Not particularly interesting job.

The second inconvenience is that the resulting setup is suitable only for running on your computer. If tomorrow you are going to have a presentation on a borrowed laptop the approach will not work. Not to mention the situation where you develop an experiment to be delivered do a broad audience with a decent deviation of computing horsepower. Such fragile solution simply does not work in practice.

### 18.2.4 Proper way of doing periodic actions on a host PC side

Fortunately, the world is full of people that need relatively accurate time measurements and lately microprocessor manufacturers are doing a fair job of keeping these people happy. For quite some time now all popular microprocessors that are finding their way into personal computers possess simple but important feature: a time-stamping counter (TSC). This is nothing more than 64-bit (it could be of different size too) counter that is incremented on each internal microprocessor clock. Therefore, if your microprocessor runs at frequency of 3.2 GHz then during each second the value of TSC increments 3,200,000,000 times (nominally, let's not forget about imperfections of quartz crystal that dictates PC clock). Despite the fact that the value of TSC increases at an enormous rate the counter hardly ever rolls over since 64-bit counter can hold so large values that even at 3 GHz frequency of incrementing it takes years to exhaust the available range.

Existence of such counter enables us to precisely measure elapsed time if only we know the rate of TSC increase per second, which differs between computers, since it depends on microprocessor clock. Fortunately, there is a way to learn this number with satisfactory accuracy and `eProDas.DLL` deduces it for you automatically. Microprocessors with built-in TSC possess dedicated instruction for reading the current TSC value; the name of instruction is RDTSC (ReaD TSC) and consequently the two eProDas functions that are related to TSC have literal RDTSC as a part of their names.

### 18.2.5 eProDas_Get_RDTSC_TickInformation

Prototype in C/C++

```
void eProDas_Get_RDTSC_TickInformation(__int64 &TicksInMicroSec,
  __int64 &TicksInMiliSec, __int64 &TicksInSec);
```

Prototype in Delphi

```
procedure eProDas_Get_RDTSC_TickInformation(var TicksInMicroSec: Int64;
  var TicksInMiliSec: Int64; var TicksInSec: Int64);
```

Prototype in VisualBasic

```
Sub eProDas_Get_RDTSC_TickInformation (ByRef TicksInMicroSec As Int64,
  ByRef TicksInMiliSec As Int64, ByRef TicksInSec As Int64)
```

This function reports increment rate of time-stamping counter TSC. Although this information can be completely presented in one integer variable it may come handy to have it presented in three (or even more) different ways. For that matter, parameters `TicksInMicroSec`, `TicksInMiliSec` and `TicksInSec` report number of TSC increments per microsecond, millisecond and second, respectively.

For example, if clock of you PC equals 3.2 GHz then the respective returned values are 3,200, 3,200,000 and 3,200,000,000. By the way, the console demo application (section 8) reports the speed of PC solely by examining one of these three variables.

### 18.2.6 ExecRDTSC

Prototype in C/C++

```
unsigned __int64 eProDas_ExecRDTSC();
```

Prototype in Delphi

```
function eProDas_ExecRDTSC(): Int64;
```

Prototype in VisualBasic

```
Function eProDas_ExecRDTSC () As Int64
```

Call this function to read the current value of TSC counter. The value is reported to you through return parameter. For C/C++ programmers there exists an inline version of this function with the name `eProDas_ExecRDTSC_Inline` by means of which you avoid the overhead of function call code. To use it, include the file `eProDasInline.h` that further relies on `windows.h` in the MS Platform SDK.

Let us demonstrate a simple example of using these functions. The following self explanatory code fragment demonstrates the way to obtain pause of one second.

```
__int64 TicksInMicroSec, TicksInMiliSec, TicksInSec;

//With the following function we learn about CPU speed.
//Call it only once and remember values in (global) variables
Get_RDTSC_TickInformation(TicksInMicroSec, TicksInMiliSec, TicksInSec);

//Reading the current value of TSC by means of which
//we time-stamp start of the pause
__int64 StartTime = eProDas_ExecRDTSC(); //in C/C++ use eProDas_ExecRDTSC_Inline if possible

//Calculate the end time-stamp. Since we need pause of one second,
//the end time-stamp equals StartTime increased for TSC increments in one second
__int64 EndTime = StartTime+TicksInSec;

//now simply wait that the actual value of TSC reaches this number
while(eProDas_ExecRDTSC()<EndTime) {} //in C/C++ use eProDas_ExecRDTSC_Inline if possible
```

**Figure 51: Pause of one second using ExecRDTSC function.**

Two things are worth to mention. First, contrary to `Sleep` function this code fragments perform the so called active pause by constantly monitoring TSC value. If you examine CPU utilization with Task Manager or some other suitable tool, you will see that CPU is occupied 100 %, which is a drawback of this method. When using function `Sleep` our program (thread) truly relinquishes its CPU time slice to other processes, which is a fair thing to do.

Despite heavier CPU utilization we need to stick with TSC method for generating time spans since data acquisition applications favour clock accuracy over fairness. When running any of such applications it is strongly recommended not to run anything else in parallel, so (neglecting kernel worker threads and interrupt handlers) there is nobody else out there that would consume the available CPU time, anyway. Alternatively, when doing long pauses you may combine both approaches. For example, your application could sleep for 950 ms and wait the remaining 50 ms as demonstrated above to achieve pause of one second.

The second thing to be stressed out is that the above code fragment is not necessarily more precise than the alternative implementation with `Sleep()` function. Multitasking OS will inevitably interrupt execution of our program to let other tasks use the CPU. Even when our data acquisition application is the only user program running at the time, there are many activities taking place within OS kernel.

So why would we use RDTSC functions, anyway? The benefit is demonstrated by the following example, where we need to read AD channel at frequency of 1 Hz.

```
__int64 TicksInMicroSec, TicksInMiliSec, TicksInSec;
Get_RDTSC_TickInformation(TicksInMicroSec, TicksInMiliSec, TicksInSec);

unsigned int ADResults;
__int64 StartTime, EndTime;

while(Exit_Condition_Is_Not_Fulfilled) {
  StartTime = eProDas_ExecRDTSC(); //time-stamping the beginning of AD reading

  //perform the activity that surely takes less than one second
  eProDas_ReadInternalADChannel(Device_Idx, ADResults);

  //Calculate the end time-stamp.
  EndTime = StartTime+TicksInSec;
  //wait that the actual value of TSC reaches this number
  while(eProDas_ExecRDTSC()<EndTime) {}
}
```

**Figure 52: Reading AD channel at frequency of 1 Hz using RDTSC (first version).**

The benefit of using RDTSC functions lays in the ability of time-stamping the events, like by using the variable `StartTime`. After start time is remembered we execute some data acquisition job as the example symbolically demonstrates. Despite the fact that we do not know exactly how much time it takes to do this job (the time may even significantly vary from one occurrence to the other) we still properly schedule consecutive events since TSC counts at constant rate independently of application activity. Reading of AD channel could take 1 ms, 100 ms or 900 ms but the calls to function `eProDas_ReadInternalADChannel` are as close to be one second apart as the underlying multitasking mechanism allows it. Contrary, using `Sleep` function would result in repetitions at time rates of 1001 ms, 1100 ms and 1900 ms.

There is still one subtlety yet to be resolved. Namely, the code fragment in the Figure 52 does not take into account the (extremely small but still existing) time that our program spends for jumping from the end of the outer while loop to its beginning and for executing the conditional statement in the inner while loop. Further, if our process is pre-empted by OS after the loop finished but before the next event is time-stamped, several milliseconds may pass before our application continues to execute. During many loop executions such small disregarded time gaps may accumulate and lead to a noticeable error. Fortunately, the solution is simple and it goes like this.

```
__int64 NextStartTime = eProDas_ExecRDTSC(); //time-stamping of first AD reading

while(Exit_Condition_Is_Not_Fulfilled) {
  //wait that the actual value of TSC reaches this number
  // (note, during the first execution of the loop the following
  //  condition is immediately fulfilled, since TSC constantly increases)
  while(eProDas_ExecRDTSC()<NextStartTime) {}

  //perform the activity that surely takes less than one second
  eProDas_ReadInternalADChannel(Device_Idx, ADResults);

  //Calculate the next start time-stamp.
  NextStartTime = NextStartTime+TicksInSec;
}
```

**Figure 53: Reading AD channel at frequency of 1 Hz using RDTSC (second version).**

Now we do not time-stamp the start of each event independently but we instead time-stamp only the beginning of the first loop. During further loop repetitions we merely increase the value of variable `NextStartTime` for the amount that TSC counter increases in one second. This way any code overhead or pre-emption gaps that were previously neglected are now taken into account, since TSC counter increments independently of these phenomena.

You probably noticed that inner while loop is moved to the beginning of outer loop in order to spend as little time between establishing the proper timing of the next event and its actual execution. In the previous example there was a jump from the end of outer loop to its beginning in between the two.

If you monitor one hour execution of the last code fragment you will count exactly 3,600±1 readings of AD channel. The tolerance of ±1 is not due to the frequency error but to the indeterminism of execution of the very last AD reading because of the multitasking effort of your OS and because of the uncertainty of the last USB packet transmission time.

The bottom line is that using RDTSC functions you can realize as time-precise data acquisition events as multitasking OS and other foes (scheduler of USB packets) permit you. You cannot do better without some sort of cheating, like moving your code into Windows kernel and running it at elevated hardware priority level. But this is another story with its own side effects… Instead, the very next chapter offers you something even better than this.

# 19 Periodic actions using dedicated eProDas features

The previous chapter describes two limiting factors (multitasking OS and USB protocol) that are degrading real-time deterministic execution of periodic activities of your choice. Both difficulties can be overcome by implementing periodic actions in a radically different way. The activities that eProDas device executes periodically must not be triggered by commands that the device receives over the USB bus. Instead, the activities should be pre-programmed into the chip and then executed at a pace that is determined by the eProDas device itself. The improvement of such approach steams from the following two paradigm shifts.

First, the clock that dictates periodic activities is synthesized by eProDas device itself. This way the remaining frequency deviations and jitter of program's execution are only due to the imperfections of the quartz crystal and the noise of the accompanying PIC's oscillator (these two imperfections limit your host PC clock as well). Neither process (thread) pre-emption on OS level nor USB packets scheduling policy influence execution of periodic activities of your eProDas device.

Second, no commands need to travel along the USB cable during the activities, since these are pre-programmed into the device. The precious and limited USB bandwidth is consumed only for real data exchange (like results of measurements and excitation data), which leads to greater data throughput. Further, communication between eProDas device and host PC during periodic activities is organized in such a way that data is grouped into as large USB packets as possible instead of e.g. transmitting each AD conversion result separately, as it is the case, when ordinary eProDas functions are used. Utilization of large USB packets leads to less USB protocol overhead and less average USB transmission delay per transferred byte.

As a drawback, the usage of dedicated eProDas periodic actions makes application coding somewhat more complicated. First, proper periodic program needs to be correctly feed into the device and some obscure decisions must be taken. Second, during periodic actions the PC application must read measurements from eProDas and send excitation data to it on a regular basis with a rather tight time tolerances (at least, if the chosen frequency of execution is fairly high). The buffering capabilities of PIC are quite limiting and if e.g. measured data is not read from the chip on time the device would inevitably have to take a break and ruin the beautiful jitterless execution of periodic activities.

## 19.1 Overall description of periodic actions

A high-level view of periodic actions is illustrated in the Figure 54. These actions are simply a set of user-specified steps that are executed whenever a period of an associated periodic clock expires ("periodic" in the name stresses that this clock dictates periodic actions, not that the clock is periodic). The period (or frequency, if you prefer) of this clock is again under the control of the user.

As presented in the bottom row of the Figure, actions during one period can be summarized into at most five parts. At the very beginning of the period, AD conversion starts, which is taken care of by PIC's hardware and therefore takes zero time (only start and not the whole AD conversion ☹). Automatic start of AD conversion is an important feature since PIC's AD converter is not exactly a fast one (if you do not cheat, as the section 15.2.2 tempts you, it takes 177 PIC's instruction cycles (14.75 μs) to complete one AD conversion if acquisition time is zero, see section 15.2.1) and therefore imposes bottleneck in many realistic scenarios of usage.

**Note 1.** If you do not need AD converter, no harm is done if you do not consume AD result later on.

**Note 2.** If internal AD module is not configured and enabled before periodic actions are initiated, AD conversion does not start at the beginning of period.

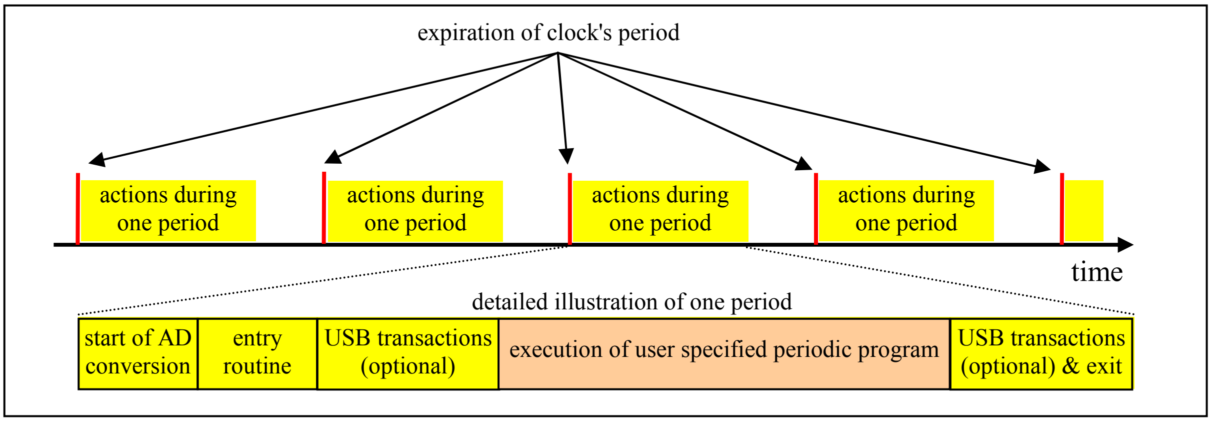

expiration of clock's period
actions during one period
actions during one period
actions during one period
actions during one period
time
detailed illustration of one period
start of AD conversion
entry routine
USB transactions (optional)
execution of user specified periodic program
USB transactions (optional) & exit

**Figure 54: High-level illustration of periodic actions.**

After AD conversion is started (or not) the entry routine starts to execute. The task of this step is to establishing a proper working environment for your periodic program (like deciding on how to handle USB transactions). The duration of entry routine is 7 PIC's instructions or 583,3 ns.

If needed (i.e. instructed by you), processing of USB transactions takes place next. The duration of this step depends on the selected mode of operation, as it is described further on.

**Note.** Special care has been taken that this step always takes the same amount of time during each period of execution in order not to introduce any unnecessary jitter into the execution of periodic program as a result of a sloppy firmware design. For example, if you select mode of operation, where this step takes 29 PIC's instruction cycles, it will always take that amount of time, regardless of the actual circumstances (like the current USB buffer must be swapped with the new one or not).

As soon as USB transactions processing (if any) is done the eProDas starts to execute a user specified periodic program (shadowed with a slightly different colour in the Figure). This is the bread-and-butter of our periodic execution. The periodic program is a sequence of steps or data-acquisition primitives that you specify in advance during configuration of periodic actions. Some examples of primitives are “read PORTx”, “write PORTx”, “read AD channel”, “write to external DA channel”...

When the program is completed, USB transactions processing may optionally take place again. After that, the period is completed, so eProDas device takes a break and it merely waits for the expiration of the next period. Then everything starts all over.

The whole point of periodic actions is now obvious. When you work with, for example, digital oscilloscope, you select the frequency at which your instrument samples certain signal that you want to observe. Similarly, when synthesizing certain signal pattern with DA converter, you send digital values of your choice to DA's input, again at a certain frequency of your selection.

How about if you were having a possibility to execute arbitrary activities at certain frequency of execution? Voila, eProDas with its periodic actions is here. If you want to sample a signal like you would do it with oscilloscope (although not that fast ☹), simply make the primitive “read AD channel” part of your periodic program. If you want to make a signal generator, use the primitive “write to external DA channel”. To realize more than one of these functionalities at the same time, simply stuff more primitives into your periodic program and there you are. By letting you specify periodic program completely from scratch and as unconstrained as possible, eProDas gives you a rich set of possibilities to implement various pieces of equipment.

## 19.2 Configuring and running periodic actions

When working with periodic actions you always follow the elaborate sequence of steps presented in the following figure.

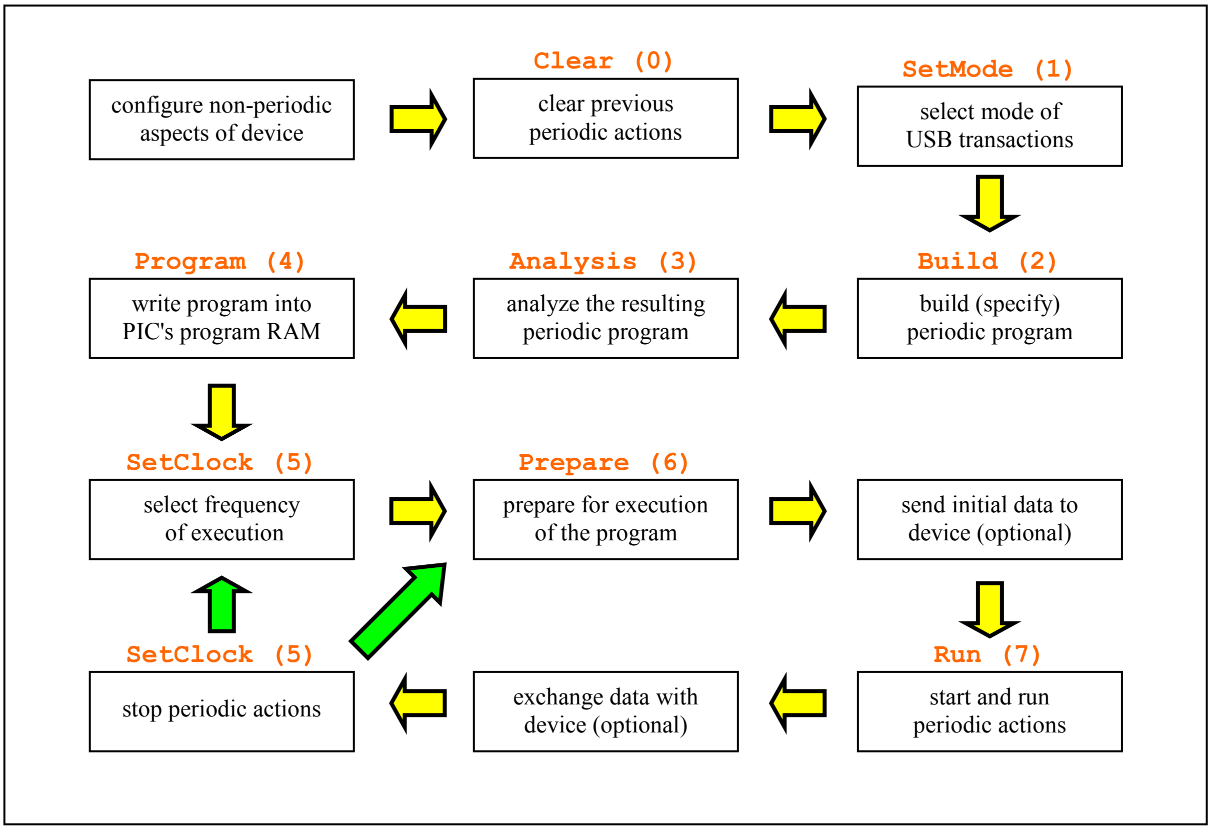

configure non-periodic aspects of device
Clear (0)
clear previous periodic actions
SetMode (1)
select mode of USB transactions
Program (4)
write program into PIC's program RAM
Analysis (3)
analyze the resulting periodic program
Build (2)
build (specify) periodic program
SetClock (5)
select frequency of execution
Prepare (6)
prepare for execution of the program
send initial data to device (optional)
SetClock (5)
stop periodic actions
exchange data with device (optional)
Run (7)
start and run periodic actions

**Figure 55: The harsh path toward periodic actions.**

The sequence looks rather scary and discouraging but we did our best to make it as easy as possible to comply with. In fact, some of the steps consist only of a trivial function call. Still, others require a substantial effort from you. The steps in the Figure 55 are not only required for proper working of periodic actions but they are forced by eProDas system, which keeps track of your moves and does not allow you to execute a certain phase of configuration without prior successful completion of the previous steps in the sequence. Here comes the function that may help you not to get lost in the jungle.

### 19.2.1 GetPeriodicConfigurationStep

Prototype in C/C++

```
int eProDas_GetPeriodicConfigurationStep(unsigned int DeviceIdx,
  unsigned int &Step);
```

Prototype in Delphi

```
function eProDas_GetPeriodicConfigurationStep(DeviceIdx: LongWord;
  var Step: LongWord): integer;
```

Prototype in VisualBasic

```
Function eProDas_GetPeriodicConfigurationStep (ByVal DeviceIdx As UInteger,
  ByRef PeriodicStep As UInteger) As Integer
```

You do not need this function at all. However, if something does not work as expected, especially at the early stage of development of your periodic application, you may find this function handy. Upon successful return the parameter Step holds the numerical value that represents the current location in the path toward periodic actions, where your application is currently standing. The possible values are listed in the following table.

| Step | associated named constant | meaning |
|---|---|---|
| 0 | eProDas_PeriodicStep_Clear | all settings regarding periodic actions are cleared |
| 1 | eProDas_PeriodicStep_SetMode | mode of USB packets handling is selected |
| 2 | eProDas_PeriodicStep_Build | you have specified at least one periodic command (primitive) |
| 3 | eProDas_PeriodicStep_Analysis | analysis of periodic program is done and no fatal errors were found |
| 4 | eProDas_PeriodicStep_Program | periodic program is written to PIC's program RAM |
| 5 | eProDas_PeriodicStep_SetClock | frequency of execution is selected |
| 6 | eProDas_PeriodicStep_Prepare | device is prepared for execution of periodic actions |
| 7 | eProDas_PeriodicStep_Run | periodic actions are running |

**Table 9: Numerical representation of periodic configuration steps.**

For each step there exists a named constant as the second column reveals. We strongly suggest you to use these in your code instead of pure numbers to reduce the chance of incompatibility problems with further eProDas versions.

The last parts of named constants equal the red-coloured annotations above the steps' descriptions in the Figure 55. For example, when you successfully complete the analysis of your periodic program, you are located in the middle rectangle of the second row on the periodic "map" and the parameter Step holds the value of eProDas_PeriodicStep_Analysis (3) upon return from the function GetPeriodicConfigurationStep.

**Note 1.** The red annotations in the Figure 55 reveal the configuration step that you reach when you successfully complete the activity in the rectangle. For example, if analysis of periodic program reveals fatal errors, the system will not promote you to the step eProDas_PeriodicStep_Analysis, but it will instead hold you back at eProDas_PeriodicStep_SetMode.

**Note 2.** If you try to subvert the system and rush with further steps without completing the ones before, eProDas will refuse your attempts with an error eProDas_Error_InvalidPeriodicSequence.

Please, excuse us such Spartan and impolite treatment, which looks like it is designed to annoy you. The truth is quite the opposite. Periodic actions are designed for speed, efficiency and data throughput. For that matter no error or consistency checking is done during the running of periodic actions. If something is not set and done well the system would crash without giving you any clue whatsoever about what went wrong. To prevent such dark scenarios we took great pain of reporting the configuration clashes immediately when they are encountered to give you clear clues about what to do to remedy the situation. We believe that such approach works much better than an ignorant silence.

Note that the annotation above the last rectangle in the Figure 55 looks like a typing error, since it repeats the same entry as it is associated with the rectangle in the row above. However, the entry is correct and it reveals that when you stop the execution of periodic actions, the system moves you to the step SetClock. This way, you can repeat the execution of the program later on without taking all the martyr's steps again. Instead, you can simply Prepare and Run the program again. Alternatively, you may change the frequency of execution as well before entering the Prepare stage. Modifications of periodic program or altering of other settings are not allowed without starting from the beginning.

## 19.3 Description of configuration steps

This section provides an overall description of steps in the Figure 55. For some of the steps a more thorough explanation follows later on when we become familiar with basic concepts that are needed to fully understand the details.

### 19.3.1 Configuration of non-periodic aspects of the device

As you can see in the Figure 55 the first thing to do is to properly configure non-periodic aspects of the device by using commands from the previous chapters (mostly 15). For example, if you intend to use port D as digital output during periodic actions you need to configure it this way. Yes, it is possible to change port's direction during periodic actions, but if you use it only as a unidirectional port, it is a waste of time to do the configuration during each period. Similarly, you configure AD converter, voltage comparators and all other device's hardware according to the requirements of your application (number of AD channels, mode of voltage comparators...) before you start working on configuration of periodic actions.

**Note.** eProDas does not force this step or care about it at all. Put it simply, if you do not configure AD converter properly, you will get meaningless AD results during periodic actions.

To be exact, configuration of non-periodic settings does not have to be done prior everything else, but we strongly recommend you to do it at least before analysis of periodic program since this analysis takes into account some of non-periodic settings and can therefore give you more comprehensive help regarding possible improvements of your periodic program.

### 19.3.2 Clearing previous periodic actions

Needless to say, people need to clean up their sins from the past before achieving new victories. Periodic actions are automatically cleared upon device's power-on and when the command `Initialize` (section 14.5.1) is executed. To clear configuration of periodic actions at any time later on, use the following function.

#### 19.3.2.1 ClearPeriodicActions

Prototype in C/C++

```
int eProDas_ClearPeriodicActions(unsigned int DeviceIdx);
```

Prototype in Delphi

```
function eProDas_ClearPeriodicActions(DeviceIdx: LongWord): integer;
```

Prototype in VisualBasic

```
Function eProDas_ClearPeriodicActions (ByVal DeviceIdx As UInteger) As Integer
```

Specifically, this command clears the previous periodic program (if any), results of analysis, settings of periodic clock and user specified mode of USB packets handling. It effectively moves you back to the rectangle `Clear(0)` in the Figure 55.

**Note 1.** You need to call this function if you want to go back to the beginning from wherever you are currently in the periodic configuration path. For example, to write new periodic program after you have run the previous one (or after any other step), there is no return without total clear.

**Note 2.** We strongly suggest you to always call this function prior to starting the work on periodic configuration. Especially, if a need arises to demonstrate your application in a public place, you cannot know in advance in what state the previous demonstrator left the eProDas device. Even better approach is to do the full initialization (section 14.5.1) at the beginning of any application.

### 19.3.3 Selection of USB transactions handling mode

In this step you configure USB transactions, which are needed for data exchange between an eProDas device and a host PC. Generally, data travels into both directions. The device sends e.g. results of measurements to the host PC whereas the latter sends excitation or signal generation data to the field.

Periodic actions could be designed in a way that there would be no need to make any decision about USB mode of operation; we could simply provide the mechanisms for sending USB packets into both directions. However, the one-size-fits-all approach is less efficient than specialized solutions since the latter can take advantage of special optimizations if the system can work with certain presumptions about your particular periodic setup.

For example, if you are only doing measurements within periodic program then you do not need any excitation data. In such cases you also do not need to waste the device's time on processing USB packets that are arriving from the host PC. Similarly, if you only do some sort of signal generation, then you do not send any measurements from the device to the host PC and consequently you do not need to waste any time on processing USB packets that travel from the device to the host PC.

These and other decisions are specified when you configure USB mode of operation. The details will have to wait for a while, so let us only reveal that this step of configuration is done by calling the to-be-described function `SetPeriodicMode` with the appropriate parameters. Upon successful return, you are promoted to the **`SetMode(1)`** level of periodic configuration.

### 19.3.4 Specifying periodic program

With this step you actually tell eProDas device what you want it to do during each period of periodic actions. For each activity that should take place, merely call the appropriate function that adds the very activity to the periodic program. Although the related functions are described further on, we are nonetheless presenting an example here to patience your curiosity about the coding style.

#### 19.3.4.1 AddPeriodicAction_TogglePin

Prototype in C/C++

```
int eProDas_AddPeriodicAction_TogglePin(unsigned int DeviceIdx,
  unsigned int PortIdx, unsigned int PinIdx);
```

Prototype in Delphi

```
function eProDas_AddPeriodicAction_TogglePin(DeviceIdx: LongWord;
  PortIdx: LongWord; PinIdx: LongWord): integer;
```

Prototype in VisualBasic

```
Function eProDas_AddPeriodicAction_TogglePin (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger, ByVal PinIdx As UInteger) As Integer
```

Whenever you want to toggle a digital state of some output pin within your periodic program, call this function and specify the pin in question by setting the parameters accordingly. For example, if your periodic program consists only of this action, then upon each periodic execution, the state of the pin would be toggled. The frequency of execution of the whole periodic program and the toggling rate would be determined by the selection of execution frequency (step **`SetClock(5)`**).

**Note.** As soon as you add the first activity to the periodic program you are promoted from the step **`SetMode(1)`** to the step **`Build(2)`**. You must specify at least one activity before proceeding with further steps; otherwise you will face the error `eProDas_Error_InvalidPeriodicSequence`.

You cannot delete the already specified activities separately. All you can do is to clear periodic actions in their entirety and start from the beginning. Fortunately, this is actually all that you need.

### 19.3.5 The Hello World periodic example (part I)

Okay, let us pause with crappy theory and start to build a simplistic periodic program. The only thing that the program is supposed to do is to toggle some pin of our choice (RC7) twice in a short time to produce a short (83.3 ns) pulse. Such activity is didactically suitable, since we do not need any data exchange between host PC and device, so we can concentrate on the basics that we are currently talking about. The first part of the program's code is presented in the Figure 56.

```
//   global definitions
 1.  const unsigned int DeviceIdx = 0;
 2.  int Status;

//   open device handle and initialize the device
 3.  Status = eProDas_OpenHandle();
 4.  if(Status != eProDas_Error_SUCCESS) { Abort_Or_Recover_Somehow(); }

 5.  Status = eProDas_Initialize(DeviceIdx);
 6.  if(Status != eProDas_Error_SUCCESS) { Abort_Or_Recover_Somehow(); }

//   non-periodic configuration of the device
 7.  Status = eProDas_ConfigurePinAsOutput(DeviceIdx, eProDas_Index_PORTC, 7, 0);
 8.  if(Status != eProDas_Error_SUCCESS) { Abort_Or_Recover_Somehow(); }

//   here we start to walk along the periodic configuration path
 9.  Status = eProDas_SetPeriodicMode(DeviceIdx, eProDas_PeriodicMode_None, 0, 0, 0);
10.  if(Status != eProDas_Error_SUCCESS) { Abort_Or_Recover_Somehow(); }

11.  Status = eProDas_AddPeriodicAction_TogglePin(DeviceIdx, eProDas_Index_PORTC, 7);
12.  if(Status != eProDas_Error_SUCCESS) { Abort_Or_Recover_Somehow(); }

13.  Status = eProDas_AddPeriodicAction_TogglePin(DeviceIdx, eProDas_Index_PORTC, 7);
14.  if(Status != eProDas_Error_SUCCESS) { Abort_Or_Recover_Somehow(); }

15.  Status = eProDas_AnalyzePeriodicActions(DeviceIdx, 1, 0, 1);
16.  if(Status != eProDas_Error_SUCCESS) { Abort_Or_Recover_Somehow(); }
```

**Figure 56: The periodic Hello World program (part I).**

The first line merely defines named constant `DeviceIdx` for the sake of greater code readability and the second line defines global `Status` variable that we use for checking the success of each operation. Lines 3 and 4 are there to open device handle, which is a must thing to do before we can do anything with eProDas system. Similarly, lines 5 and 6 are there to stress the good practice of always initializing device before starting to work with it; this step also clears all settings regarding periodic actions so we know for sure that we are now at the step **`Clear(0)`** in the Figure 55.

**Note.** We check the success of each operation by examining the `Status` variable after the respective function call. This treating stresses the good and the only correct coding practice.

The true happening starts at line 7, where the non-periodic device configuration begins. In our case this step consists only of configuring pin RC7 as digital output (section 15.1.3), so that the pulses can truly appear on the pin. At the same time we also select the initial logic state of the pin to be low (0).

Line 9 selects mode of USB transactions, which we have merely mentioned so far. Nonetheless, you should have no trouble grasping the idea that the first parameter of the function `SetPeriodicMode` selects no USB transactions at all, since we do not need them. We are now at the step **`SetMode(1)`**.

To produce the pulse we need to toggle the pin twice. Two toggling actions are added to the periodic program in the lines 11 and 13. As soon as the line 11 executes, we are promoted to the step **`Build(2)`**, where we stay as long as we do not do anything else but add further periodic activities to our program.

Line 15 announces the next required step in the periodic configuration path: the mighty analysis.

### 19.3.6 Analysis of periodic program

Despite the fact that periodic programs are simple sequences of user-specified activities there is a need for their thorough analysis. The first information that we would like to have is the maximal possible frequency of operation. The figure depends on the duration of the program and the longer the periodic program executes the lower is the permissible frequency of its repetition. If the program takes, say 200 µs to execute, we cannot possibly expect to repeat it at a greater frequency than 1/(200 µs) = 5 kHz.

But how long does it take to execute? We could simply tell you the execution time of each periodic action plus the eProDas's overheads and then demand from you to sum up all these numbers to get the execution time. Needless to say, you would not be happy with such treatment. Instead, there it is a special function `AnalyzePeriodicActions` that does the job for you.

Further, this function helps you exploit internal AD converter in an efficient way. As already stated, we are talking about relatively slow AD and waiting for completion of AD conversion is a nuisance waste of time. Since AD conversion starts automatically upon expiration of period (see the bottom line of the Figure 54), you would want to do as much other tasks as possible before consuming AD result. eProDas already gives you the choice of processing USB transactions during AD conversion, but this step takes much less than 177 PIC's instructions that AD converter needs to chew its potato. So, you want to stuff as much other activities as possible at the beginning of your periodic program to fulfil the time gap that AD converter dictates. The function `AnalyzePeriodicActions` pays attention to AD activity and gives you hints about how to optimize the arrangement of your periodic instructions.

Last but not least the results of analysis are needed for proper functioning of eProDas periodic subsystem. Besides determining the maximal permissible frequency this function calculates optimal sizes of USB packets for data exchange during periodic actions and some other parameters that are needed for proper execution. Therefore, even if you do not want to take a look at the results of the analysis you are still obliged to analyze you program by calling the following function.

#### 19.3.6.1 AnalyzePeriodicActions

Prototype in C/C++
```
int eProDas_AnalyzePeriodicActions(unsigned int DeviceIdx,
  unsigned int TextualReport, unsigned int Disassembly, unsigned int SaveToFile);
```

Prototype in Delphi
```
function eProDas_AnalyzePeriodicActions(DeviceIdx: LongWord;
  TextualReport: LongWord; Disassembly: LongWord; SaveToFile: LongWord): integer;
```

Prototype in VisualBasic
```
Function eProDas_AnalyzePeriodicActions (ByVal DeviceIdx As UInteger,
  ByVal TextualReport As UInteger, ByVal Disassembly As UInteger,
  ByVal SaveToFile As UInteger) As Integer
```

The results of the analysis are remembered internally by `eProDas.DLL` in order to properly execute periodic actions at a later time. In addition, this function can generate textual report (plain ASCII text) of analysis that contains program flow calculations and remarks about errors and warnings that it encounters during analysis. If you want this textual report to be generated, specify a non-zero value for the parameter `TextualReport`.

The textual report lists all periodic activities in a human-readable way, i.e. it provides descriptive annotations of them. Hard core PIC hackers may be interested in the actual disassembly code behind the scenes, which is included in the report by setting parameter `Disassembly` to a non-zero value. If you are lucky enough to belong to the group of normal human beings and you do not know what assembler is, make sure that you leave this parameter at zero; you are not missing anything.

The easiest way to read the generated textual report is to set the parameter `SaveToFile` to a non-zero value. The report appears in a file `$$$_eProDas_PeriodicAnalysis_AutoGenFile_$$$.txt` that is saved into the current application's folder. Use your favourite text editor to read it. Alternatively, it is possible to export the ASCII text to the buffer of your choice which is described further on.

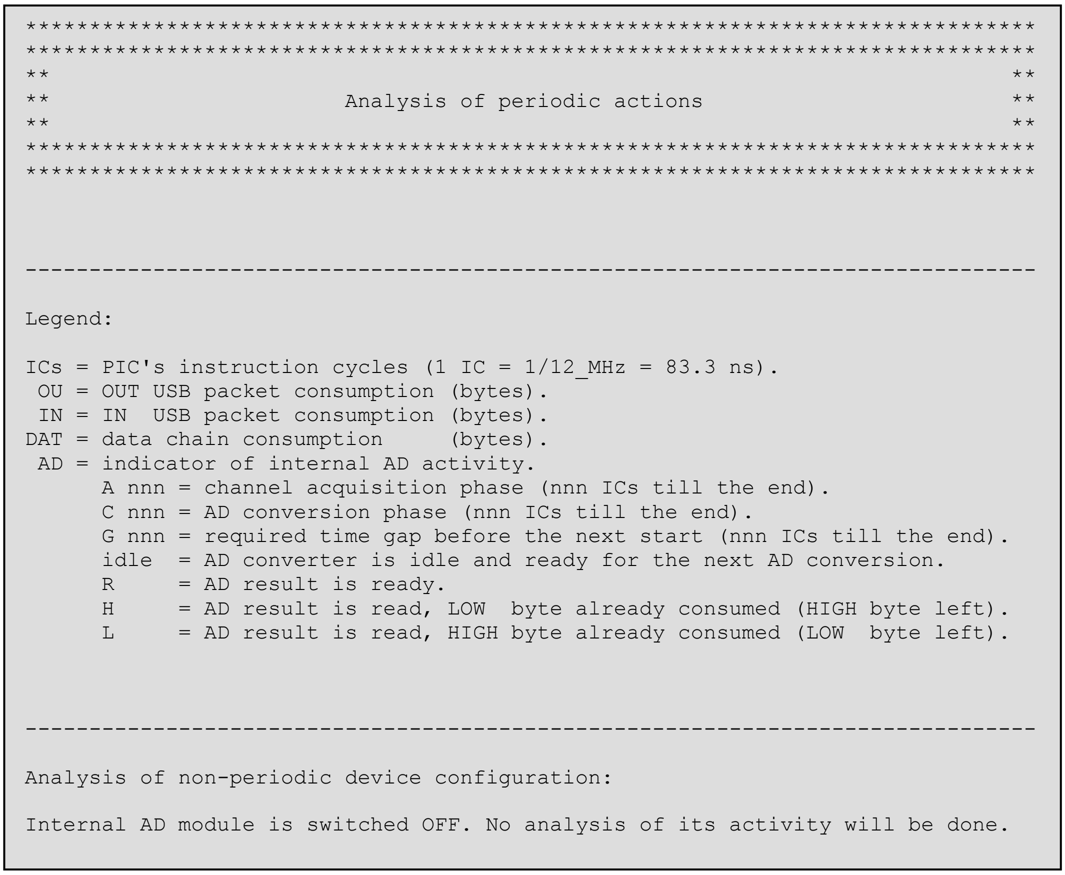
The intended usage of the function `AnalyzePeriodicActions` is as follows. When you develop certain periodic application for the first time, generate textual report and examine it. Correct errors and other issues that are reported and repeat the analysis (probably several times). When everything is polished to a satisfactory extent, you can stop generating textual reports and call the function merely to satisfy eProDas capriciousness. If periodic program and non-periodic configuration of eProDas device do not change in the future then the generated report will remain unchanged too.

**Note.** If the analysis of periodic program reveals any fatal errors, the function will return with error `eProDas_Error_InvalidPeriodicProgram` and the system will not let you proceed with further steps of periodic configuration until you remedy the situation and successfully complete the analysis.

If no fatal errors are discovered during the analysis, you are promoted to the step **`Analysis(3)`**.

### 19.3.6.2 Analysis of the Hello World program

Let us see what the analyzer has to say about our exemplar program in the Figure 56. By examining line 15 of the listing we can deduce that generation of textual report without disassembly was instructed, which is a preferred way for the majority of eProDas application developers. The beginning of the report is presented in the following figure.



```
********************************************************************************
********************************************************************************
**                                                                            **
**                        Analysis of periodic actions                        **
**                                                                            **
********************************************************************************
********************************************************************************




--------------------------------------------------------------------------------

Legend:

ICs = PIC's instruction cycles (1 IC = 1/12_MHz = 83.3 ns).
 OU = OUT USB packet consumption (bytes).
 IN = IN  USB packet consumption (bytes).
DAT = data chain consumption     (bytes).
 AD = indicator of internal AD activity.
      A nnn = channel acquisition phase (nnn ICs till the end).
      C nnn = AD conversion phase (nnn ICs till the end).
      G nnn = required time gap before the next start (nnn ICs till the end).
      idle  = AD converter is idle and ready for the next AD conversion.
      R     = AD result is ready.
      H     = AD result is read, LOW  byte already consumed (HIGH byte left).
      L     = AD result is read, HIGH byte already consumed (LOW  byte left).



--------------------------------------------------------------------------------

Analysis of non-periodic device configuration:

Internal AD module is switched OFF. No analysis of its activity will be done.
```

**Figure 57: The analysis report of the Hello World program (part I).**

The major part of the presented output consists of a legend that helps us navigate the material that follows (in the next figure). However, note the last line. We have not configured and enabled internal AD module prior to requesting the analysis. eProDas notices that and simply warns us that it will not help us optimize AD's usage. This is fine for our case, since we really do not intend to do any AD conversions.  However, if we had utilized AD in our periodic program, we would surely like to configure the module before the analysis starts, to get all the help that we can.

The continuation of the report follows in the next figure. This material is much more interesting, since it mentions everything that is going to happen during each period of execution.

```
********************************************************************************
********************************************************************************
**                                                                            **
**           Listing of activities during one period of execution             **
**                                                                            **
********************************************************************************
********************************************************************************



No.    Label         Description of periodic action       ICs  OU IN DAT   AD
---- ---------- ----------------------------------------- ---- -- -- --- -------

auto            Entry routine                                7

   1            Toggle pin RC7                               1

   2            Toggle pin RC7                               1

auto            Exit from periodic actions                   3
```

**Figure 58: The analysis report of the Hello World program (part II).**

Each activity occupies one row of the listing. The columns from left to right have the following meaning. Column "No." counts user-specified activities, however if the activity is provided by eProDas automatically, the entry "auto" is placed in the column instead of the sequence number. The third column "Description…" provides a human-understandable description of the listed activity. The next column "ICs" summarizes the duration of the activity in PIC's instruction cycles (83.3 ns). The remaining columns are going to be ignored for now.

The following figure repeats the block diagram of one period of execution, which we have already had the chance to observe in the bottom part of the Figure 54.

| | | | | |
|---|---|---|---|---|
| start of AD conversion | entry routine | USB transactions (optional) | execution of user specified periodic program | USB transactions (optional) & exit |

**Figure 59: Block diagram of one period of execution (repeated).**

Let us compare how these two pictures fit together. According to the Figure 59 the first thing that happens is the start of AD conversion. Although we inspect the Figure 58 with great care we are not able to find any evidence of this activity. The reason is simple: AD converter is not enabled and therefore AD conversions do not start (the last line in the Figure 57 warned us, didn't it).

Then the entry routine follows in the Figure 59 and we feel a bit relaxed since this is indeed mentioned in the first line of the listing. The activity number is "auto" since entry routine is provided by eProDas automatically. Under the column "ICs" we can see that the duration of this step is 7 PIC's instructions.

By following the doctrine of the Figure 59, the handling of USB transactions could happen next, but this is not the case as the Figure 58 assures us. Such outcome was expected, since the line 9 of our program (Figure 56) demanded no USB transactions.

Now we have reached the meaning of the program's life. The next two activities are just what we want: toggling of pin RC7 twice. Each of these two activities takes one PIC's instruction to execute.

The exit routine, which is again needed for proper working of the system, follows our periodic program and is provided by eProDas automatically. This piece of code takes 3 IC's to execute.

The following figure provides the last part of the report, where the summary of the analysis is provided. This material is also important for the application's developer.

```
********************************************************************************
********************************************************************************
**                                                                            **
**               USB packets calculation and summary of analysis              **
**                                                                            **
********************************************************************************
********************************************************************************



Mode of packets processing: none



--------------------------------------------------------------------------------



Total user-specified periodic commands: 2

ESTIMATED duration of the whole periodic program: 1.00 us (12 ICs)

ESTIMATED maximal possible frequency: 1.000 MHz



--------------------------------------------------------------------------------






********************************************************************************
********************************************************************************
**                                                                            **
**                     The end of periodic analysis report                    **
**                                                                            **
********************************************************************************
********************************************************************************
```

**Figure 60: The analysis report of the Hello World program (part III).**

Mode of USB packets processing is none, as we already know, so this is not big news. We also know that we added two activities to our periodic program. However, the total duration of all activities together, which is kindly provided in the next line, would be a bit complicated to calculate by hand. As it turns out the entry routine, the two our actions and the exit routine together take 12 PIC's instructions to execute, which takes exactly 1 µs of our precious time. Accordingly, the maximal permissible frequency of execution is 1 MHz as the last useful line of the report reveals.

Note the word “ESTIMATED” at the beginning of the two lines that report the duration and maximal permissible frequency. eProDas does its best to calculate the reported figures correctly by disassembling the actual code to be executed. But nonetheless, this is only a simple analyzer and not a full-blown PIC simulator. Although not in simple cases as the Hello World program is, it is possible or even likely that the calculations are not accurate or sensible. If there are conditional statements (even implicit ones) in your program, it is obvious that the analyzer cannot know whether the conditions are fulfilled or not. Further, if some activity depends on the outside asynchronous events, the analyzer cannot possibly know in advance about the timings of events. For that matter, the system gives you the opportunity to object these figures in you believe that they are wrong. Just stick with us to see how.

The positive side of this report is that it does not contain any signs of encountered errors, which means that eProDas is satisfied with the outcome of the analysis and it will let us run the program. We are now promoted to the step `Analysis(3)`.

### 19.3.7 Writing the program to PIC’s program RAM

Okay, we have got the correct program at hand. So far, eProDas device has not been touched by our periodic configuration. The periodic program that is a result of everything described so far is merely remembered internally by `eProDas.DLL`. Now, the time has come to write this program to the PIC’s program RAM where the activities can truly execute. Here comes the function for the task.

#### 19.3.7.1 ProgramPeriodicActions

Prototype in C/C++

```
int eProDas_ProgramPeriodicActions(unsigned int DeviceIdx);
```

Prototype in Delphi

```
function eProDas_ProgramPeriodicActions(DeviceIdx: LongWord): integer;
```

Prototype in VisualBasic

```
Function eProDas_ProgramPeriodicActions (ByVal DeviceIdx As UInteger) As Integer
```

The definition of the function is really relaxing since no special parameters need to be set. You may be wondering why this step is not done automatically by eProDas upon successful completion of the analysis. The only reason that your intervention is required is that PIC’s program RAM is of a non-volatile FLASH type. This memory cannot be reprogrammed for indefinite number of times without being destroyed. According to datasheet a typical (not worst case) chip can be reprogrammed for about 100,000 times before it is burnt out, which is a fairly large number but finite nonetheless. For that matter, eProDas does not initiate any programming of the chip if you do not explicitly order it. It is possible that your periodic program does not contain any errors but you are still not satisfied with it and you continue polishing it for ten more times before arriving at the ultimate solution. In this situation we have saved ten burnings and programming of the memory by waiting for your command to transfer the end result to the chip.

After successful programming you are at the step `Program(4)`. Since the memory is not volatile the periodic program remains on the chip even when the power is off. The discussed function always checks the current contents of the program RAM before erasing and re-programming it. If your program is already on the chip, this function will save the chip by not touching it.

**Note 1.** Although you may be sure that the chip already contains the proper program, you must nonetheless call this function, since eProDas will stubbornly refuse to go on without first checking that your chip is correctly programmed. The function `ProgramPeriodicActions` spares your chip as much as possible, so there is no need for bad conscious in such cases, since PIC’s program RAM is merely read but not written to if not strictly necessary.

**Note 2.** Whenever you change periodic program to be executed, the reprogramming phase must truly take place. If you change periodic program several times a day, you will inevitably shorten PIC's life. For example, by assuming that 100,000 reprogramming phases are possible, your chip will last for about 274 years if you reprogram it once per day, but it will manage to live only for 27.4 years if you program it ten times per day. It is like with human beings, if you live rave, your life will be shorter but more plentiful ☺.

### 19.3.8 Setting clock of execution

When eProDas system analyzes your periodic program it learns (or it thinks so at least) about the maximal possible frequency of program execution, as it was presented in the Figure 60. Now it allows you to select the desired frequency of periodic actions (to be promoted to the step `SetClock(5)`), which has to be less than or equal to the calculated maximal value. You can use the following two functions for setting periodic clock (parameter `Forced` is described later on, set it to zero for now).

#### 19.3.8.1 SetPeriodicClockFrequency

Prototype in C/C++

```
int eProDas_SetPeriodicClockFrequency(unsigned int DeviceIdx,
  unsigned int FrequencyHz, unsigned int Forced);
```

Prototype in Delphi

```
function eProDas_SetPeriodicClockFrequency(DeviceIdx: LongWord;
  FrequencyHz: LongWord; Forced: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_SetPeriodicClockFrequency (ByVal DeviceIdx As UInteger,
  ByVal FrequencyHz As UInteger, ByVal Forced As UInteger) As Integer
```

With the function `SetPeriodicClockFrequency` you set periodic clock in a usual way by specifying the desired frequency in Hertz. Simply assign value of, say, 1,000 to the parameter `FrequencyHz` when you want to execute periodic actions at frequency of 1 kHz.

#### 19.3.8.2 SetPeriodicClockPeriod

Prototype in C/C++

```
int eProDas_SetPeriodicClockPeriod(unsigned int DeviceIdx,
  unsigned int MicroSeconds, unsigned int MiliSeconds, unsigned int Seconds,
  unsigned int Minutes, unsigned int Hours);
```

Prototype in Delphi

```
function eProDas_SetPeriodicClockPeriod(DeviceIdx: LongWord;
  MicroSeconds: LongWord; MiliSeconds: LongWord; Seconds: LongWord;
  Minutes: LongWord; Hours: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_SetPeriodicClockPeriod (ByVal DeviceIdx As UInteger,
  ByVal MicroSeconds As UInteger, ByVal MiliSeconds As UInteger,
  ByVal Seconds As UInteger, ByVal Minutes As UInteger, ByVal Hours As UInteger,
  ByVal Forced As UInteger) As Integer
```

In cases of extremely small frequencies, like sub 1 Hz, specifying frequency is rather non-intuitive and a better way is to specify the period between the starts of two adjacent periodic actions. This is what the function `SetPeriodicClockPeriod` is about. By setting values of self descriptive parameters you specify the desired period which is the sum of all the values. This way it is extremely simple to express the desires like: make one sample in 5 hours, 23 minutes and 45 seconds.

In order not to lose the possibility of generating demands by some simplified program, individual parameters are not checked for overflowing their usual intervals. It is perfectly legal to specify period of 0 hours, 214 minutes and 763 seconds, which is equivalent to a more aesthetically pleasant combination of 3 hours, 47 minutes and 43 seconds.

#### 19.3.8.3 Limitations of periodic clock

There is much to be written about periodic clock and how to set it, so we devote the entire section 19.7 to this topic. Nonetheless, the most important constraints should be stated now, i.e. before you touch the previous two functions for the first time.

**First.** By default eProDas does not allow you to set periodic clock to a higher value than the maximal estimated frequency of execution (periodic analysis, section 19.3.6). If you think that this dumb machine is wrong and that your program could be executed at a higher frequency set the parameter `Forced` to a non-zero value and eProDas will silently comply with your orders (up to the unrealistic frequency of 12 MHz). If you exaggerate and your program cannot be run at the demanded frequency, nothing wrong will happen, since the system is designed to be resistant to such attempts. Only your program will execute slower than you expect. Note also that execution may not be jitter-less, since USB transactions, which are processed at the end of periodic program (Figure 59), are designed for fast exit rather than for uniform execution delay. When two periods of execution bumps together, the second one will start to execute as soon as the previous one exits. Therefore, if execution time of USB transactions varies from period to period, the frequency cannot be uniform either.

**Second.** Periodic clock is obtained by dividing PIC's 12 MHz instruction clock by an integer divider. Frequencies that do not divide 12 MHz evenly cannot be set. For, example, by dividing 12 MHz by 12 we obtain frequency of 1 MHz, whereas division by 13 gives 923,076.9 Hz. Nothing in-between is possible to synthesize. If your demand cannot be fulfilled exactly, both functions for setting the clock try to find as good approximation as possible. You may want to check the outcome with the function `GetPeriodicClock` (section 19.7.2) to be on a safe side.

**Third.** The maximal allowed period is about one day, so the frequency cannot be arbitrarily low even if it divides 12 MHz evenly.

### 19.3.9 Preparing periodic actions

This step is straightforward. Simply call function `PreparePeriodicActions`, which has no parameters at all (except unavoidable `DeviceIdx`).

#### 19.3.9.1 PreparePeriodicActions

Prototype in C/C++

```
int eProDas_PreparePeriodicActions(unsigned int DeviceIdx);
```

Prototype in Delphi

```
function eProDas_PreparePeriodicActions(DeviceIdx: LongWord): integer;
```

Prototype in VisualBasic

```
Function eProDas_PreparePeriodicActions (ByVal DeviceIdx As UInteger) As Integer
```

With this function you signal eProDas system that you have finished programming and tweaking periodic actions, so now the system can get ready for their execution. As a part of its job the function `PrepareForPeriodicActions` initializes and prepares the data transfer subsystem for its task; all USB buffers of eProDas device are flushed and reinitialized.

If you have not cheated, the function will promote you to the step **`Prepare(6)`**.

**Note.** If you change your mind after this step and you do not want to execute your periodic program, you can only `ClearPeriodicActions`, which takes care of putting the alerted device back to the idle state.

## 19.3.10 Running periodic actions

Everything is in place, so we can trigger the thing by calling the function `StartPeriodicActions`, which fortunately ☺ has only one simple parameter other than the nuisance `DeviceIdx`.

### 19.3.10.1 StartPeriodicActions

Prototype in C/C++

```
int eProDas_StartPeriodicActions(unsigned int DeviceIdx,
  unsigned int RaisePriority);
```

Prototype in Delphi

```
function eProDas_StartPeriodicActions(DeviceIdx: LongWord;
  RaisePriority: LongWord): integer;
```

Prototype in VisualBasic

```
Function eProDas_StartPeriodicActions (ByVal DeviceIdx As UInteger,
  ByVal RaisePriority As UInteger) As Integer
```

If the function succeeds, you are not only promoted to the step **Run(7)**, but your masterpiece becomes alive, too!

When periodic actions are running eProDas device cannot execute any "ordinary" commands. As far as the host PC is concerned three activities are being done exclusively:

1. eProDas device transmits its packets (e.g. results of measurements…) to the host PC,
2. eProDas device receives packets (e.g. excitation data…) from the host PC,
3. eProDas device waits for the (yet to be described) `StopPeriodicActions` command.

Trying to execute anything else results in an error `eProDas_Error_InvalidDuringPeriodicActions` but your periodic program is not aborted in such cases.

During periodic actions your application is responsible for collecting data (if any) that device generates and for sending the host PC generated data (if any) to the device. Failing to do so in proper time intervals would exhaust a rather limited buffering capabilities of the device and result in buffer overrun (when device-generated packets are not pulled out of device fast enough) or buffer underrun (in the case of host PC-generated packets not being sent to the device on time).

When either buffer overrun or underrun happens, the device pauses the execution and waits for the missing packet(s) before it continues running. The already generated data (results of measurements) are never overwritten by the new data, which is usually more favourable thing to do than preserving the unaltered frequency of execution. Also, the device counts such events internally and it reports them to you upon termination of periodic actions.

If you specify a non-zero value of the parameter `RaisePriority`, the application's priority is raised to the `real-time` priority class (if you are not familiar with workings of Windows scheduler just try to grasp the idea) in order to give the time critical nature of periodic actions as much chance as possible to transfer packets on time. If the user account, under which your application is running, does not possess the necessary privilege for such priority boost, this step is silently skipped (usually application is still elevated to the `high` priority class, which is still better than the `normal` privilege level).

You have to decide for yourself whether the rising of application's priority is needed or not. For high throughput cases, this may be well worth it; otherwise you lose more than you win. Namely, while a high priority application (thread) is occupying CPU, Windows become irresponsive; your mouse pointer follows a mouse with annoying delay… As a very imperfect rule of thumb: rise application's priority, if the periodic program demands average delivery of packets on a millisecond scale or faster. However, if it is enough to deliver one packet per, say, 10 ms or more, the normal priority may be all that you need. The best way to decide is to experiment and see if priority boost makes a difference.

## 19.3.11 Stopping periodic actions

When your application decides that it is time to stop periodic actions, it calls the following function.

### 19.3.11.1 StopPeriodicActions

Prototype in C/C++

```
int eProDas_StopPeriodicActions(unsigned int DeviceIdx, unsigned int FinishSent,
  unsigned int FinishReceived, unsigned char *PostSendBuffer,
  unsigned int &SendViolations, unsigned int &ReceiveViolations);
```

Prototype in Delphi

```
function eProDas_StopPeriodicActions(DeviceIdx: LongWord; FinishSent: LongWord;
  FinishReceived: LongWord; PostOUTBuffervar: PByte; var SendViolations: LongWord;
  var ReceiveViolations: LongWord): integer;
```

Prototype in VisualBasic

```
Function eProDas_StopPeriodicActions (ByVal DeviceIdx As UInteger,
  ByVal FinishSent As UInteger, ByVal FinishReceived As UInteger,
  ByRef PostOUTBuffervar As Byte, ByRef SendViolations As UInteger,
  ByRef ReceiveViolations As UInteger) As Integer
```

You certainly expected this step to be fairly straightforward and the myriad of function parameters undoubtedly surprised you. Well, it turns out that putting the beast on sleep is not that easy at all. The details will have to wait for a while, but we can reveal you that the attempt will succeed if you set the first two parameters to zero and the third parameter to null-pointer. The last two parameters are set by the function to the internally counted number of buffer underruns and buffer overruns, respectively. If periodic actions were running normally, these two parameters are set to zero.

When periodic actions are stopped, you are transferred to the step **SetClock(5)**, so that you can prepare and run the program again or you can change the frequency of execution and then prepare and run.

## 19.3.12 The Hello World periodic example (part II)

Now it's time to take a deep breath and do something funny. We left our exemplar Hello World program from the Figure 56 at the point of successful completion of its analysis. The story unfolds in the Figure 61.

The programming of the PIC is done in the line 17, where we are promoted to the step **Program(4)**.

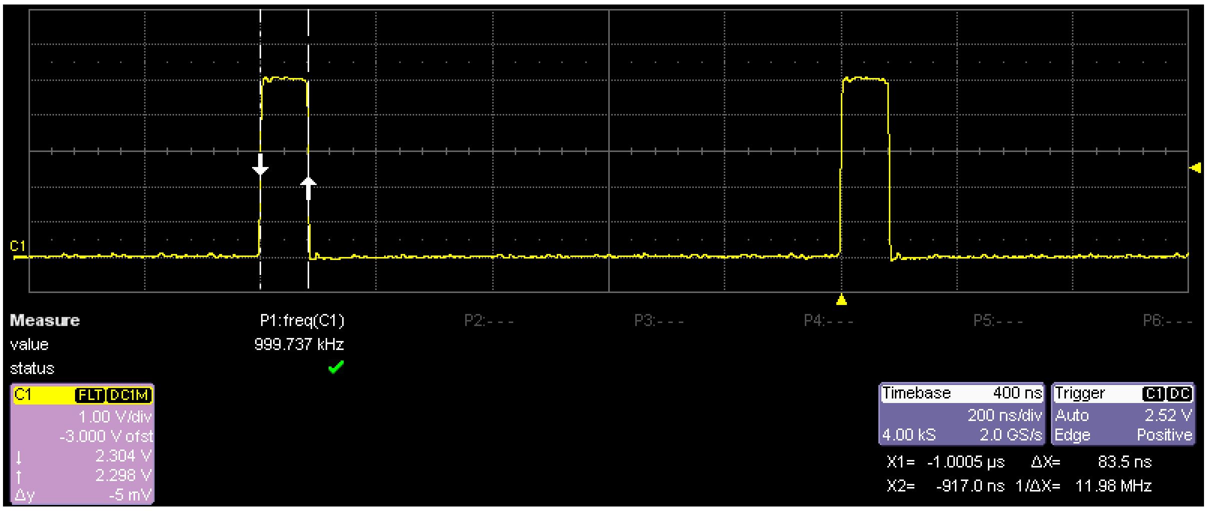
Now we have to decide about the frequency of execution. The analysis report offered us the maximal frequency of 1 MHz (Figure 60) and for greedy developers as we are the maximum is the only choice. This frequency is selected in the line 20 with a little help from the line 19. Now we stand on the stair **SetClock(5)**.

The line 22 prepares the system for the race and promotes as to the step **Prepare(6)**. One more function call in the line 24 and we are up and **Run(7)**-ing. **Champagne everyone!**

```
//  write program to the device's program RAM
Status = eProDas_ProgramPeriodicActions(DeviceIdx);
if(Status != eProDas_Error_SUCCESS) { Abort_Or_Recover_Somehow(); }

//  select frequency of execution
unsigned int DesiredFrequency = 1000000;
Status = eProDas_SetPeriodicClockFrequency(DeviceIdx, DesiredFrequency, 0);
if(Status != eProDas_Error_SUCCESS) { Abort_Or_Recover_Somehow(); }

//  prepare periodic actions
Status = eProDas_PreparePeriodicActions(DeviceIdx);
if(Status != eProDas_Error_SUCCESS) { Abort_Or_Recover_Somehow(); }

//  start the beast
Status = eProDas_StartPeriodicActions(DeviceIdx, 0);
if(Status != eProDas_Error_SUCCESS) { Abort_Or_Recover_Somehow(); }

//  keep it running for as long as you want
Somehow_pause_until_periodic_program_has_been_running_for_long_enough();

//  stop the beast
unsigned int SendViols, ReceiveViols;
Status = eProDas_StopPeriodicActions(DeviceIdx, 0, 0, 0, SendViols, ReceiveViols);
```

**Figure 61: The periodic Hello World program (part II).**

From now on our periodic program executes one million times per second. You can check this by hooking your favourite oscilloscope to the pin RC7 and examine the generated voltage waveform. You should see something like the Figure 62 shows.



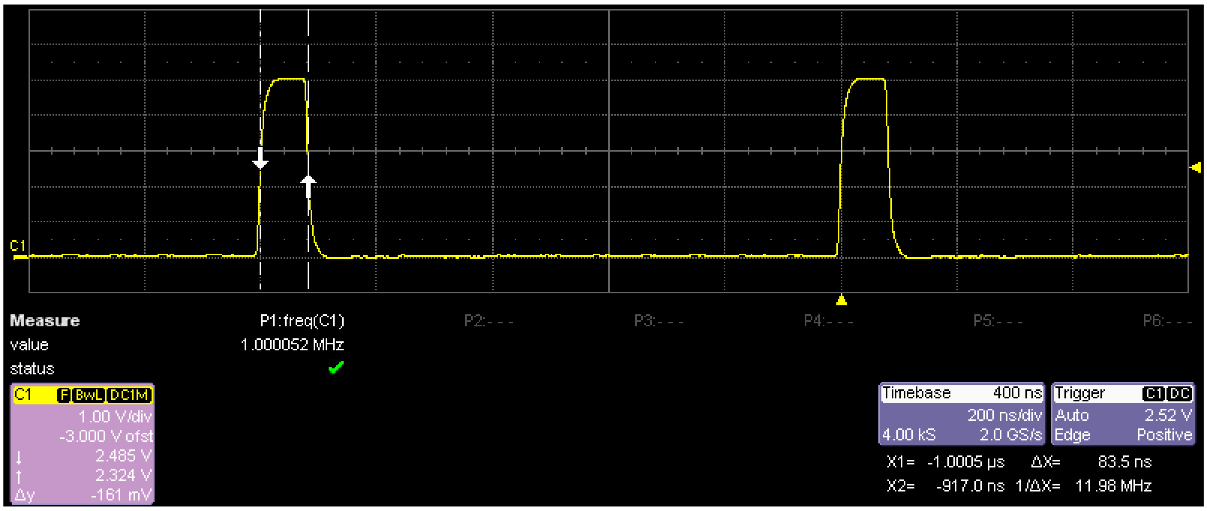
```
Measure   P1:freq(C1)   P2:- - -   P3:- - -   P4:- - -   P5:- - -   P6:- - -
value     999.737 kHz
status    ✓

C1  FLT DC1M                     Timebase      400 ns   Trigger   C1 DC
   1.00 V/div                               200 ns/div  Auto      2.52 V
   -3.000 V ofst                 4.00 kS    2.0 GS/s    Edge      Positive
↓  2.304 V                       X1=  -1.0005 µs   ΔX=     83.5 ns
↑  2.298 V                       X2=   -917.0 ns  1/ΔX=   11.98 MHz
Δy -5 mV
```

**Figure 62: A voltage waveform on the pin RC7 during execution of Hello World program.**

At this point a few things are worth to mention. First, you can see from the frequency measurement result in the Figure 62 that the mean frequency of generated pulses in not exactly 1 MHz as expected but it instead equals 999.737 kHz. If we assume that the measurement is completely exact (but nothing is perfect…) we can calculate the relative error of the generated frequency with the following expression: (1 MHz – 999.737 kHz) / (1 MHz) = 2.63 x $10^{-4}$. This level of accuracy is to be expected from the cheap old quartz that generates PIC's clock and consequently this is the level of performance that you can expect from a fairly cheap data acquisition system.

Second, although this is not a textbook on electronics we feel the need to comment the clean and proper shape of pulses in the Figure 62. The frequency of 1 MHz is high enough that some of the often overlooked phenomena come to the play. The frequency content of the signal is much higher than 1 MHz since we are not generating a sine wave of that frequency but sharp rectangle pulses, so the bandwidth of several tenths of megahertz is needed to feed such signal through the system without significant distortion (the required bandwidth is in fact dictated by the rise or fall time of the voltage and not by the frequency of pulses).

The Figure 62 was obtained with oscilloscope that claims bandwidth of 200 MHz. Please compare the shape with the one in the Figure 63, where the oscilloscope's bandwidth was reduced to 20 MHz.



```
Measure     P1:freq(C1)    P2:- - -   P3:- - -   P4:- - -   P5:- - -   P6:- - -
value       1.000052 MHz
status      ✓

C1  F BwL DC1M                          Timebase      400 ns   Trigger     C1 DC
        1.00 V/div                                 200 ns/div  Auto        2.52 V
      -3.000 V ofst                     4.00 kS     2.0 GS/s   Edge      Positive
↓       2.485 V                         X1=  -1.0005 µs    ΔX=      83.5 ns
↑       2.324 V                         X2=    -917.0 ns  1/ΔX=   11.98 MHz
Δy      -161 mV
```

**Figure 63: Observed waveform on the pin RC7 with reduced oscilloscope bandwidth.**

The pulses look quite distorted but this is just an illusion due to the limitations of the equipment (or its setup in this case). The measured frequency of pulses is now 1.000052 MHz, which gives us relative error of 5.2 x $10^{-5}$. The figure look better but this a pure coincident. Due to noise and imperfections of the quartz crystal each clock period is of slightly different length, which we call jitter. One good measurement does not mean much, and a proper approach to estimate clock's imperfections is to measure the longer term instability and deviations of its period.

Let as return to the shape of the pulses. The point is that you need equipment of a sufficient quality and you also have to use it properly to observe the true shape of the signals within this frequency range. As we have seen, the bandwidth of oscilloscope has to be sufficiently wide but the probes are extremely important as well. To obtain the waveform in Figure 62 we utilized a probe of a decent quality with a short ground wire for the task. Please, examine the encircled area in the Figure 64.

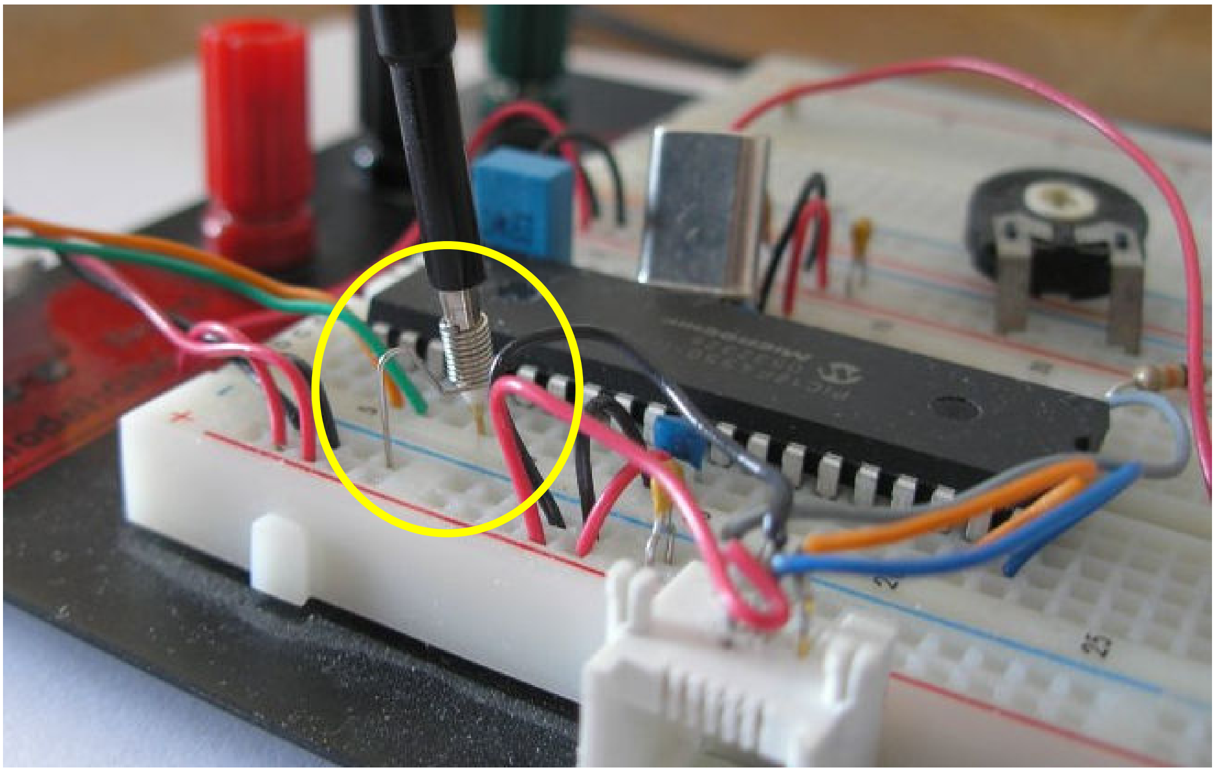


Figure 64: Using scope probe with a short ground wire.

To see why you need to use probes with adequate specifications let us connect the scope to the eProDas system with two ordinary wires as presented in the following figure.

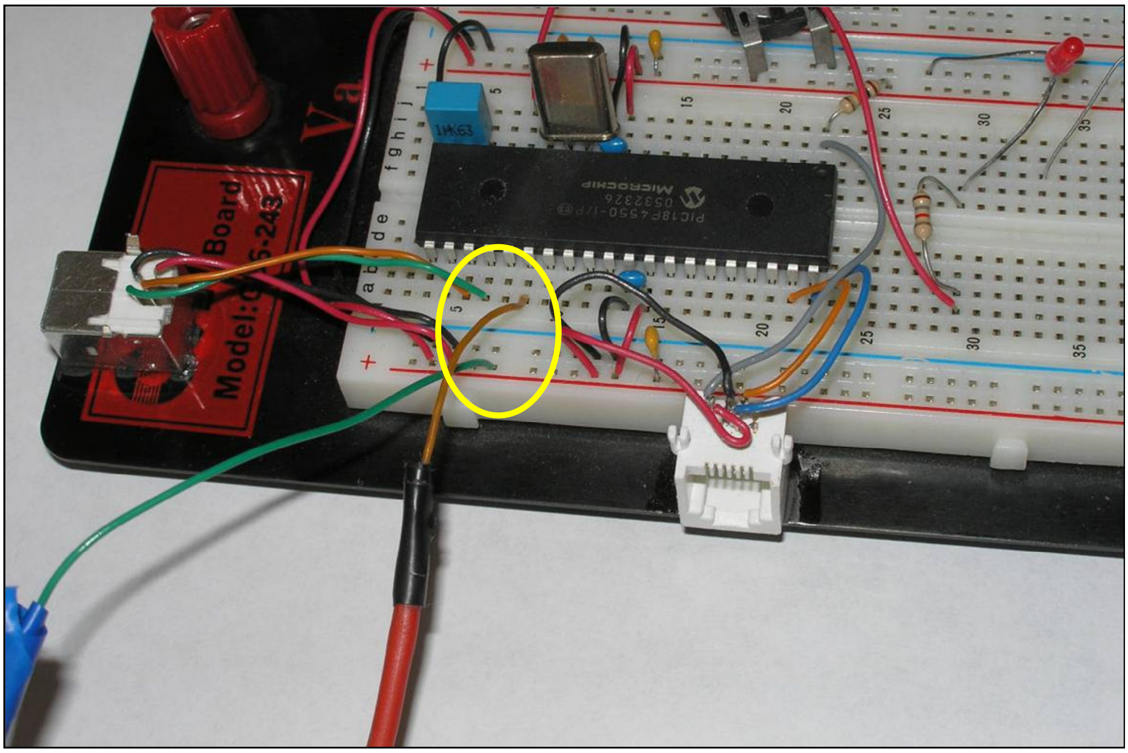
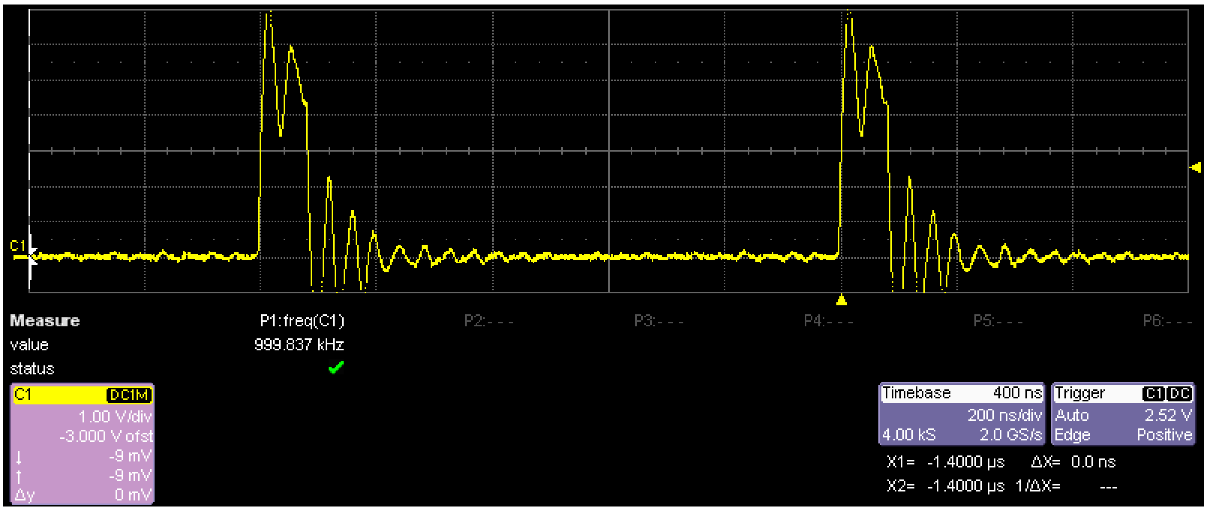


Figure 65: Replacing scope probe with ordinary wires.

Without altering anything else the picture on scope's display changes from the one in the Figure 62 to the one in the Figure 66 (bandwidth is back to 200 MHz, again).



Measure P1:freq(C1) P2:- - - P3:- - - P4:- - - P5:- - - P6:- - -
value 999.837 kHz
status ✓

C1 DC1M
1.00 V/div
-3.000 V ofst
↓ -9 mV
↑ -9 mV
Δy 0 mV

Timebase 400 ns
200 ns/div
4.00 kS 2.0 GS/s

Trigger C1 DC
Auto 2.52 V
Edge Positive

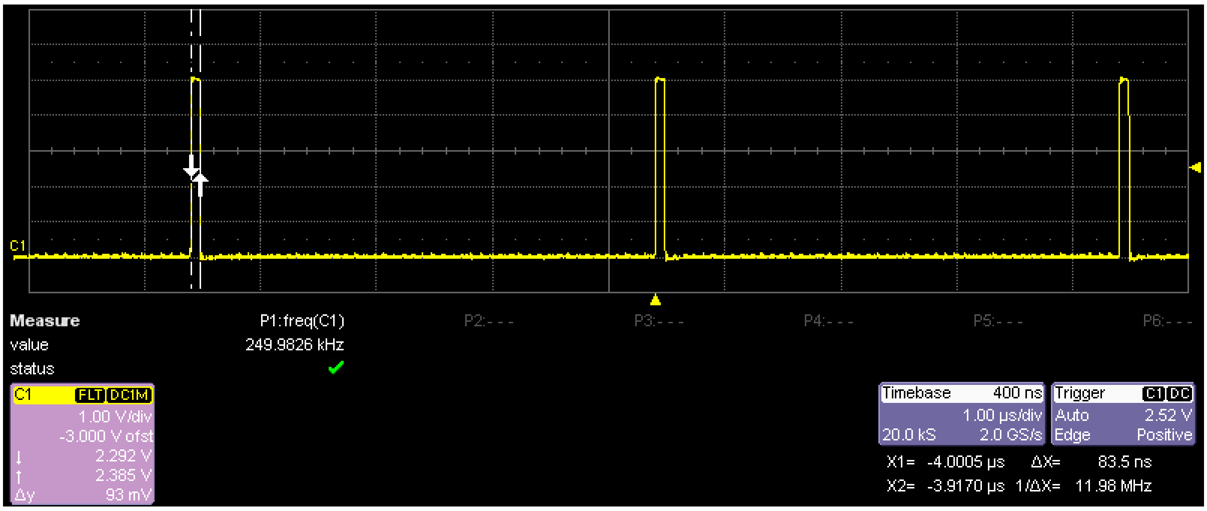
X1= -1.4000 µs ΔX= 0.0 ns
X2= -1.4000 µs 1/ΔX= ---

**Figure 66: Distorted signal waveform.**

It is hard not to notice the characteristics ringing whenever the signal abruptly changes the voltage level. The bottom line is: eProDas system can generate signals for you, but it is your responsibility to preserve these signals as undistorted as necessary for your particular task. The signal on pin RC7 will be probably used as a digital signal and digital systems are rather tolerant to such phenomena (but the one in the Figure 66 is really worrying) at least until the ringing is so excessive that the signal crosses digital threshold more than once during its transition.

A more problematic situation arises when such signal is used, for example, as a clock for pipelined AD converter. In this case the ringing and distortion (and jitter, of course) may degrade precision of AD conversions. Therefore, if you are unfamiliar with these phenomena we recommend you to search for literature on keywords like: lines, transmission lines, line termination, AC termination, etc. and you will manage to equip yourself with enough knowledge to escape the discussed trap.

Let us show one more snapshot of our oscilloscope's display. This time the probe is back but the frequency of execution is lowered from 1 MHz to 250 kHz. The outcome is visible in the Figure 67.



Measure P1:freq(C1) P2:- - - P3:- - - P4:- - - P5:- - - P6:- - -
value 249.9826 kHz
status ✓

C1 FLT DC1M
1.00 V/div
-3.000 V ofst
↓ 2.292 V
↑ 2.385 V
Δy 93 mV

Timebase 400 ns
1.00 µs/div
20.0 kS 2.0 GS/s

Trigger C1 DC
Auto 2.52 V
Edge Positive

X1= -4.0005 µs ΔX= 83.5 ns
X2= -3.9170 µs 1/ΔX= 11.98 MHz

**Figure 67: A voltage waveform on the pin RC7 at 250 kHz.**

With this demonstration we want to stress the following fact, which you already know. Although the frequency is lowered (note the time axis scale change from 200 ns/div to 1 μs/div), the duration of the pulses is still 83.3 ns (the measured duration is 83.5 ns). This means that our periodic program always executes at the same speed (i.e. the two toggling commands are always 83.3 ns apart) from the beginning to the end. The frequency of periodic clock determines the time between the starts of two adjacent executions. Therefore, the first fronts of the pulses in the Figure 62 are 1 μs apart since the program executes at a frequency of 1 MHz, whereas the first fronts of the pulses in the Figure 67 are 4 μs apart since the program executes at a frequency of 250 kHz.

Okay, we are nearly at the end with our Hello World demonstration. All that remains is to properly stop the periodic activity as the line 28 in the Figure 61 does. Upon stopping the periodic activity, the system returns `SendViolations` and `ReceiveViolations` statuses. Although we do not deal with USB packets at all and we know in advance that there cannot possibly be any violations whatsoever we still need to provide storage space for these two returned values which is the task of line 27.

As soon as the line 28 executes the pattern on the oscilloscope vanishes and there remains a straight and a slightly noisy line...

## 19.4 USB packets and data exchange

Our Hello World program does not communicate with host PC at all, but this is a rare exception. In general cases data travel in USB packets bidirectionally between eProDas device and host PC. According to the USB terminology, which is strictly PC centric, IN direction denotes transfer of packets from device to the host PC whereas OUT direction means the opposite. Therefore, IN packets carry results of measurements (e.g. AD results), read values from ports... whereas OUT packets transfer excitation data to be written to ports, voltage reference, potentially presented DA converters...

Since the underlying medium for transfer is USB bus, eProDas must take into account its properties and peculiarities in order to squeeze as much performance out of the system as possible. The most annoying property for high performance data-acquisition system is the already mentioned latency of isolated packet delivery. USB bus tries to keep high *average* but not *instant* data rates, so it does not bother to give each packet its full acceleration; but if there happens to be four or five packets in a queue the average latency per transferred byte will be significantly lower, since the whole bunch of them will be delivered in a single USB frame (1 ms) if feasible.

As a result, under the hood eProDas device contains more than one IN and more than one OUT buffer for smooth transmission of packets. Currently, there are 6 buffers for each direction, which is as many of them as the amount of data RAM on PIC chip permits us to implement and we would be glad to raise the number of buffers if only that were feasible. The more buffers we have the more tolerant is the system about unpredictable delays of USB transfers.

In order for these buffers to be of any use it is recommended that your application keeps them as busy as possible at all times during periodic actions in order to minimize the risk of buffer overruns (when device wants to write data to new IN buffer, but none is empty) or buffer underruns (when device wants to read data from new OUT buffer, but none contains the needed data). At least this should be the case with high-speed, high-data-rate periodic actions; however if your particular setup requires delivery of one packet per several milliseconds or even less frequent, the requirements are much less stringent.

### 19.4.1 Data exchange and periodic program

Although we are constantly repeating the boring word “packet”, the mechanism of data exchange may be better grasped if we think in terms of contiguous IN and OUT data pipes (or FIFO buffers, data streams) that happen to be physically divided into packets for technical reasons.

Your periodic program is not concerned about USB packets at all. Instead it only knows about the next byte in the IN pipe to be written to with the new data and the next byte in the OUT pipe to be read. Everything else is just a background framework for making data exchange along the USB cable feasible and efficient.

Let us introduce periodic program in the Figure 68 to explain the concept. We are neither concerned about the usefulness of the program nor we give you a full code example together with non-periodic and periodic configuration (analysis, clock’s setting, preparation…), which is necessary to make the example truly run.

```
 1 eProDas_AddPeriodicAction_ReadPort(DeviceIdx, eProDas_Index_PORTD);
 2 eProDas_AddPeriodicAction_TogglePin(DeviceIdx, SomePort, SomePin);
 3 eProDas_AddPeriodicAction_ReadPort(DeviceIdx, eProDas_Index_PORTE);
 4 eProDas_AddPeriodicAction_WritePort(DeviceIdx, eProDas_Index_PORTB);
 5 eProDas_AddPeriodicAction_ReadInternalAD(DeviceIdx);
 6 eProDas_AddPeriodicAction_WritePort(DeviceIdx, eProDas_Index_PORTA);
 7 eProDas_AddPeriodicAction_WritePort(DeviceIdx, eProDas_Index_PORTC);
 8 eProDas_AddPeriodicAction_TogglePin(DeviceIdx, SomePort, SomePin);
 9 eProDas_AddPeriodicAction_WritePort(DeviceIdx, eProDas_Index_PORTE);
10 eProDas_AddPeriodicAction_ReadInternalComparators(DeviceIdx);
```

**Figure 68: Sample dummy periodic program.**

We took the liberty of using a couple of periodic commands, that we have not described yet, but you should have no problem understanding the concept despite our impoliteness.

- Periodic action `ReadPort` in line 1 reads the current state of some port (D in this case) and writes the result into the IN pipe.

- Line 2 toggles state of some pin, which does nothing with USB packets (oops, pipes).

- Line 3 again reads some port (now it is E) and writes its current state into the IN pipe.

- Periodic action `WritePort` in line 4 reads the next byte from OUT pipe and writes it to the selected port (B in this case).

- Line 5 reads the result of AD conversion on a currently acquired AD channel and writes it into the IN pipe. This adds two bytes to the IN pipe, since AD result occupies more than eight bits.

- Lines 6 and 7 each read one byte from OUT pipe and write them to the ports of our choice (A and C, respectively).

- Line 8 again toggles state of some pin, which does nothing with USB pipes.

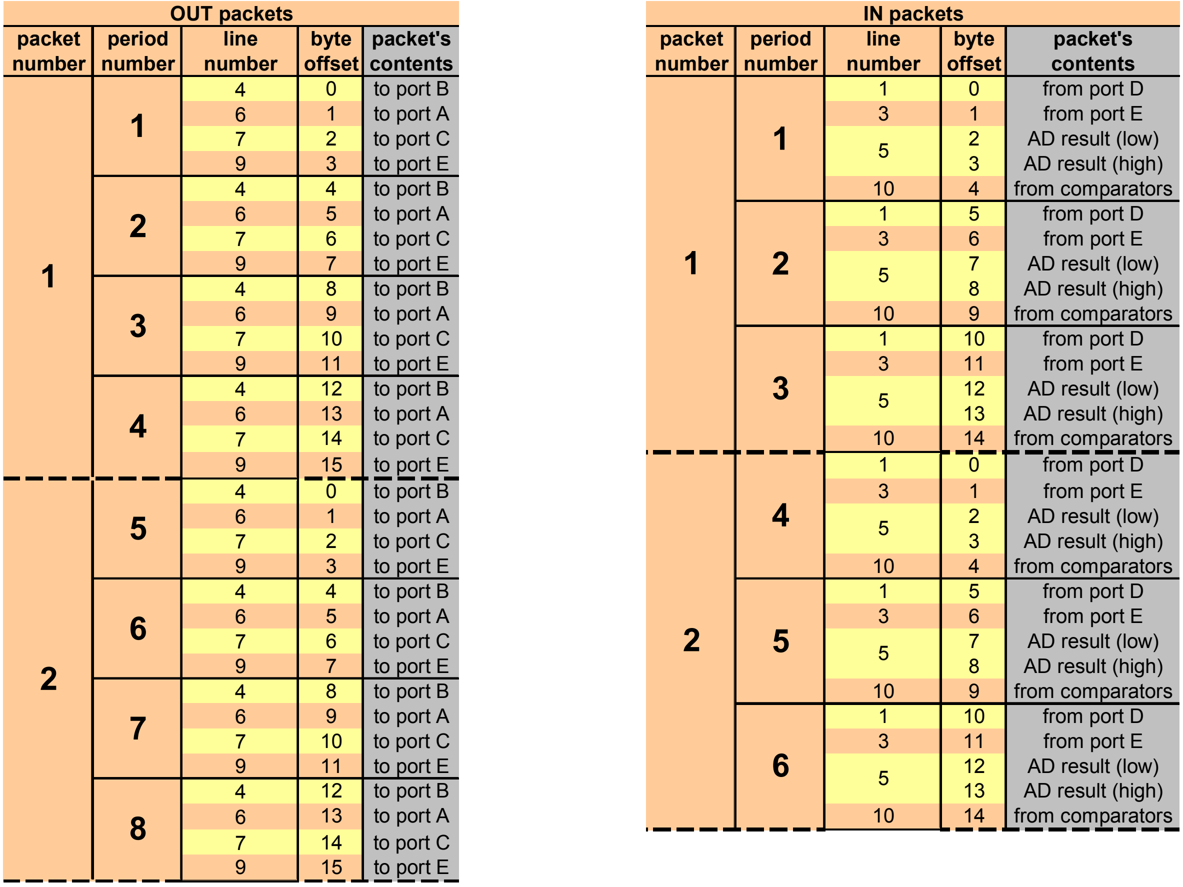
- Line 9 reads one byte from OUT pipe and writes it to the port E.

- Finally, line 10 reads state of comparators and writes the result into the IN pipe.

*On average*, for each period of execution our application (host PC) provides 4 bytes of OUT data that eProDas device consumes. Also, *on average*, host PC also receives 5 bytes of IN data that eProDas device generates along the way. The words *on average* stress that data exchange does not happen after each period of execution but rather when certain buffer is full (IN) or empty (OUT). OUT and IN pipes are independent of each other so nothing is wrong if periodic program generates different amount of data than it consumes.

Now, let us imagine that eProDas cuts OUT pipe into packet once per 4 periods of execution. Similarly, let it cut IN pipe into packet once per 3 periods of execution. Actually, you prescribe for each pipe after how many periods it is cut into the new packet. Again, nothing is wrong if OUT and IN cutting instants differ.

For the program in the Figure 68 the contents of OUT and IN packets are presented in the left and right parts of the Figure 69, respectively. The greyed columns describe the actual contents and all the rest serves for annotation.



OUT packets

| packet number | period number | line number | byte offset | packet's contents |
|---|---|---|---|---|
| 1 | 1 | 4 | 0 | to port B |
| | | 6 | 1 | to port A |
| | | 7 | 2 | to port C |
| | | 9 | 3 | to port E |
| | 2 | 4 | 4 | to port B |
| | | 6 | 5 | to port A |
| | | 7 | 6 | to port C |
| | | 9 | 7 | to port E |
| | 3 | 4 | 8 | to port B |
| | | 6 | 9 | to port A |
| | | 7 | 10 | to port C |
| | | 9 | 11 | to port E |
| | 4 | 4 | 12 | to port B |
| | | 6 | 13 | to port A |
| | | 7 | 14 | to port C |
| | | 9 | 15 | to port E |
| 2 | 5 | 4 | 0 | to port B |
| | | 6 | 1 | to port A |
| | | 7 | 2 | to port C |
| | | 9 | 3 | to port E |
| | 6 | 4 | 4 | to port B |
| | | 6 | 5 | to port A |
| | | 7 | 6 | to port C |
| | | 9 | 7 | to port E |
| | 7 | 4 | 8 | to port B |
| | | 6 | 9 | to port A |
| | | 7 | 10 | to port C |
| | | 9 | 11 | to port E |
| | 8 | 4 | 12 | to port B |
| | | 6 | 13 | to port A |
| | | 7 | 14 | to port C |
| | | 9 | 15 | to port E |

IN packets

| packet number | period number | line number | byte offset | packet's contents |
|---|---|---|---|---|
| 1 | 1 | 1 | 0 | from port D |
| | | 3 | 1 | from port E |
| | | 5 | 2 | AD result (low) |
| | | | 3 | AD result (high) |
| | | 10 | 4 | from comparators |
| | 2 | 1 | 5 | from port D |
| | | 3 | 6 | from port E |
| | | 5 | 7 | AD result (low) |
| | | | 8 | AD result (high) |
| | | 10 | 9 | from comparators |
| | 3 | 1 | 10 | from port D |
| | | 3 | 11 | from port E |
| | | 5 | 12 | AD result (low) |
| | | | 13 | AD result (high) |
| | | 10 | 14 | from comparators |
| 2 | 4 | 1 | 0 | from port D |
| | | 3 | 1 | from port E |
| | | 5 | 2 | AD result (low) |
| | | | 3 | AD result (high) |
| | | 10 | 4 | from comparators |
| | 5 | 1 | 5 | from port D |
| | | 3 | 6 | from port E |
| | | 5 | 7 | AD result (low) |
| | | | 8 | AD result (high) |
| | | 10 | 9 | from comparators |
| | 6 | 1 | 10 | from port D |
| | | 3 | 11 | from port E |
| | | 5 | 12 | AD result (low) |
| | | | 13 | AD result (high) |
| | | 10 | 14 | from comparators |

**Figure 69: USB packets organization for sample dummy periodic program.**

Before we `StartPeriodicActions`, we must send at least one OUT packet to the device (but we can send as many as 6 of them to fill in all 6 OUT buffers). If this is not complied with, buffer underrun would happen at the very beginning of the execution. Contrary, IN packets are empty at the beginning of execution and they are waiting for periodic program to fill them with data that is going to be generated during the execution.

Now we start running our periodic program by executing the function `StartPeriodicActions`. The happening in the Figure 54 (page 166) becomes alive and we are at the moment when the very first period of periodic actions begins to execute.

The first instruction reads the current state of port's D pins and writes the result into the first available byte of IN packet. Since IN packet is empty at the moment, the result is written at offset 0, as it is shown on the right part of the Figure 69.

The next periodic command toggles `SomePin` which does nothing with packets so this action can be observed by oscilloscope but not by examining the contents of the packets.

Line 3 reads port E and writes the result into the next available IN byte slot, which now happens to be the one at offset 1.

Line 4 reads the next available byte of OUT packet and writes the contents to port B. Since this is the first reading from the OUT packet, the byte at offset 0 is read.

The next command reads the result of the last AD conversion. First, the least significant byte of the result is written to the IN packet at the next available byte (offset 2) and after that the most significant byte is written to the byte at offset 3.

Lines 6 and 7 read the next byte of OUT packet (offsets 1 and 2) and write the contents to ports A and C, respectively.

The next periodic command again toggles `SomePin` which does nothing with packets.

Line 9 reads the contents of OUT packet at offset 3 and writes whatever is there to the port E.

As the last step, the state of comparators is written into the IN packet at offset 4. After that, the periodic interpreter gets some rest until the beginning of the next period starts.

If you managed to follow the lengthy explanation without falling into sleep you have summed up the gross result that 4 bytes of OUT packet have been consumed and 5 bytes of IN packet have been filled in with generated data.

Since we said that the OUT pipe is cut into packet after each four periods, the system knows that the currently used OUT buffer must not be released yet because there are 12 more useful bytes to be consumed during the following three periods of execution. Similarly, the IN pipe is cut into the packet after each three periods; the IN buffer must accept 10 more bytes during two additional execution cycles, before the resulting packet gets ready to be sent to the host PC.

Sooner or latter the next period of execution begins and everything repeats from the scratch. The first command again reads the contents of port D and stores the results into the next available byte of IN packet, which happens to be at offset 5. The second line does not touch packets at all. The third line writes the contents of port E into the IN buffer at offset 6. The fourth line reads the next byte from OUT packet (offset 4) and writes it to port B. In the line 5 AD result is stored at offsets 7 and 8 of the IN buffer. The next two lines transfer OUT packet's contents at offsets 5 and 6 to ports A and C, respectively. After pin toggling, the byte at offset 7 of OUT packet is written to port E and the state of comparators is written to the IN buffer at offset 9.

The consumed amount of OUT packet is now 8 bytes whereas the current size of IN packet is 10 bytes. Not enough periods has passed by yet so eProDas device keeps IN packet for itself and it does not send it to the host PC. Also, it does not release the currently used OUT buffer, so no new data can arrive to it from the host PC.

After running periodic program for the third time we consume 12 bytes of OUT packet whereas the IN packet grew to the size of 15 bytes. Since three periods have passed by the system now cuts IN pipe into the packet. The buffer that contains the 15 bytes of generated data gets ready for transmission. At the same time an internal IN pointer is reinitialized to the start address of the next IN buffer. Therefore, one out of 6 IN buffers is now full and ready for delivery, whereas 5 other IN buffers are empty. During the next three periods of execution, the second IN buffer is going to be full and ready for delivery, whereas 4 of them will be still empty. The story goes on again and again.

When your application accepts the data in some IN buffer, the very buffer becomes empty and ready to be filled in when the time comes. If all 6 buffers are full and your application has not managed to pull any data out of chip, then buffer overrun happens. eProDas will not overwrite any data but it will instead delay further execution of periodic program until your application finally manages to accept at least one IN buffer.

The handling of OUT packets is analogous. After the first four executions of periodic program, the system cuts OUT pipe. The first OUT buffer is declared empty and ready to accept new data from the host PC. At the same time an internal OUT pointer is reinitialized to the start address of the next OUT buffer. Under the presumption that our application filled in all 6 buffers before start, one out of 6 OUT buffers is now empty and ready for reception of new data, whereas 5 other OUT buffers are still full. During the next four periods of execution, the second OUT buffer is going to be emptied and ready for new data, whereas 4 of them will still be full.

When your application sends new data, the proper OUT buffer gets filled in. However, if all 6 buffers are empty and your application has not managed to send additional data to of chip, then buffer underrun happens. eProDas will not use the empty buffer but it will instead delay further execution of periodic program until your application finally manages to send some data to the device.

### 19.4.1.1 Summary of periodic USB transfers

The main point of the previous lengthy description is that your periodic program is not concerned about packets at all. It merely knows about the next IN and OUT bytes to be written to or read from, respectively. Periodic program does not know and it should not know with which one of the six OUT and IN buffers it is working currently.

eProDas framework assures that there cannot be any buffer overruns or buffer underruns *during* the *execution* of periodic program, since packets' sizes are always integer multiples of one period's consumption amount. The splitting of pipes into packets happens before or after the execution of the whole period, as the following figure illustrates.

| | | | | |
|---|---|---|---|---|
| start of AD conversion | entry routine | USB transactions (optional) | execution of user specified periodic program | USB transactions (optional) & exit |

**Figure 70: Block diagram of one period of execution (with emphasis on USB transactions).**

The Figure 70 repeats the block diagram of one period of execution; this time the emphasis is on USB transactions before and after the user's periodic program. The two shaded blocks in the figure perform the swapping of the buffers and create an illusion that physical buffers are indefinite pipes. During execution of the user's program (in between the two shaded rectangles) there is always room for new data to be written to and there is always enough data available for reading as needed for one period.

Application on a host PC also sees eProDas device as two independent pipes that just happen to deliver the data in chunks of certain length. The application cannot request e.g. reading of the fourth IN buffer or writing to the third OUT buffer. It can only read/write from/to the next IN/OUT buffer.

## 19.5 Toward running of the sample dummy program

Let us try to make the sample dummy program from Figure 68 alive. The first part of the effort is illustrated with the following code fragment.

```
//  global definitions
 1. const unsigned int DeviceIdx = 0;
 2. const unsigned int ADInputs = 1;
 3. const unsigned int Acq = 0;
 4. const unsigned int Vref = 0;
 5. const unsigned int CompMode = 1;
 6. int Status;

 // open device's handle
 7. Status = eProDas_OpenHandle();

 // non-periodic configuration of the device
 8. Status = eProDas_Initialize(DeviceIdx);
 9. Status = eProDas_ConfigureInternalAD(DeviceIdx, ADInputs, Acq, Vref, Vref);
10. Status = eProDas_ConfigureInternalComparators(DeviceIdx, CompMode, 0, 0, 0);

 // periodic configuration path
11. Status = eProDas_SetPeriodicMode(DeviceIdx, eProDas_PeriodicMode_None, 0,0,0);

12. Status = eProDas_AddPeriodicAction_ReadPort(DeviceIdx, eProDas_Index_PORTD);
13. Status = eProDas_AddPeriodicAction_TogglePin(DeviceIdx, eProDas_Index_PORTA,5);
14. Status = eProDas_AddPeriodicAction_ReadPort(DeviceIdx, eProDas_Index_PORTE);
15. Status = eProDas_AddPeriodicAction_WritePort(DeviceIdx, eProDas_Index_PORTB);
17. Status = eProDas_AddPeriodicAction_ReadInternalAD(DeviceIdx);
18. Status = eProDas_AddPeriodicAction_WritePort(DeviceIdx, eProDas_Index_PORTA);
19. Status = eProDas_AddPeriodicAction_WritePort(DeviceIdx, eProDas_Index_PORTC);
20. Status = eProDas_AddPeriodicAction_TogglePin(DeviceIdx, eProDas_Index_PORTA,5);
21. Status = eProDas_AddPeriodicAction_WritePort(DeviceIdx, eProDas_Index_PORTE);
22. Status = eProDas_AddPeriodicAction_ReadInternalComparators(DeviceIdx);

23. Status = eProDas_AnalyzePeriodicActions(DeviceIdx, 1, 0, 1);
```

**Figure 71: Code for running the sample dummy periodic program (part I, try I).**

We have omitted error checking to make the code snapshot more conscious, but this is not what you should do, and we at least symbolize our non-ignorance by assigning each error code to the `Status` variable during the program flow.

Let us see, what we have got here. Lines from 1 to 5 merely introduce some named constants for better code clarity and line 6 declares the well known `Status` variable. Lines 7 and 8 are familiar as well.

Line 9 is a bit more interesting, since it configures internal AD module, which is going to be accessed during periodic actions. The actual configuration of the module does not matter for this example, though. Similarly, line 10 configures internal comparators, which are also monitored by our periodic program.

Periodic configuration path begins in line 11, where we set mode of USB packets handling. We selected no data transfer whatsoever, since we have not studied the function `SetPeriodicMode` yet and this mode of operation is all that we know about at the moment.

Lines from 12 to 22 are there to build our periodic program from the Figure 68. Finally, line 23 instructs the required analysis of the program. Since this is the first time that we work on the sample dummy program, we are certainly interested in examining the outcomes of the analysis and fortunately, the first part of the report is presented in the next figure.

```
*******************************************************************************
*******************************************************************************
**                                                                           **
**                        Analysis of periodic actions                       **
**                                                                           **
*******************************************************************************
*******************************************************************************




-------------------------------------------------------------------------------

Legend:

ICs = PIC's instruction cycles (1 IC = 1/12_MHz = 83.3 ns).
 OU = OUT USB packet consumption (bytes).
 IN = IN  USB packet consumption (bytes).
DAT = data chain consumption     (bytes).
 AD = indicator of internal AD activity.
      A nnn = channel acquisition phase (nnn ICs till the end).
      C nnn = AD conversion phase (nnn ICs till the end).
      G nnn = required time gap before the next start (nnn ICs till the end).
      idle  = AD converter is idle and ready for the next AD conversion.
      R     = AD result is ready.
      H     = AD result is read, LOW  byte already consumed (HIGH byte left).
      L     = AD result is read, HIGH byte already consumed (LOW  byte left).



-------------------------------------------------------------------------------

Analysis of non-periodic device configuration:

Internal AD module is switched ON and configured properly.
AD time intervals: acquisition: 0 ICs, conversion: 177 ICs, gap: 32 ICs.
```

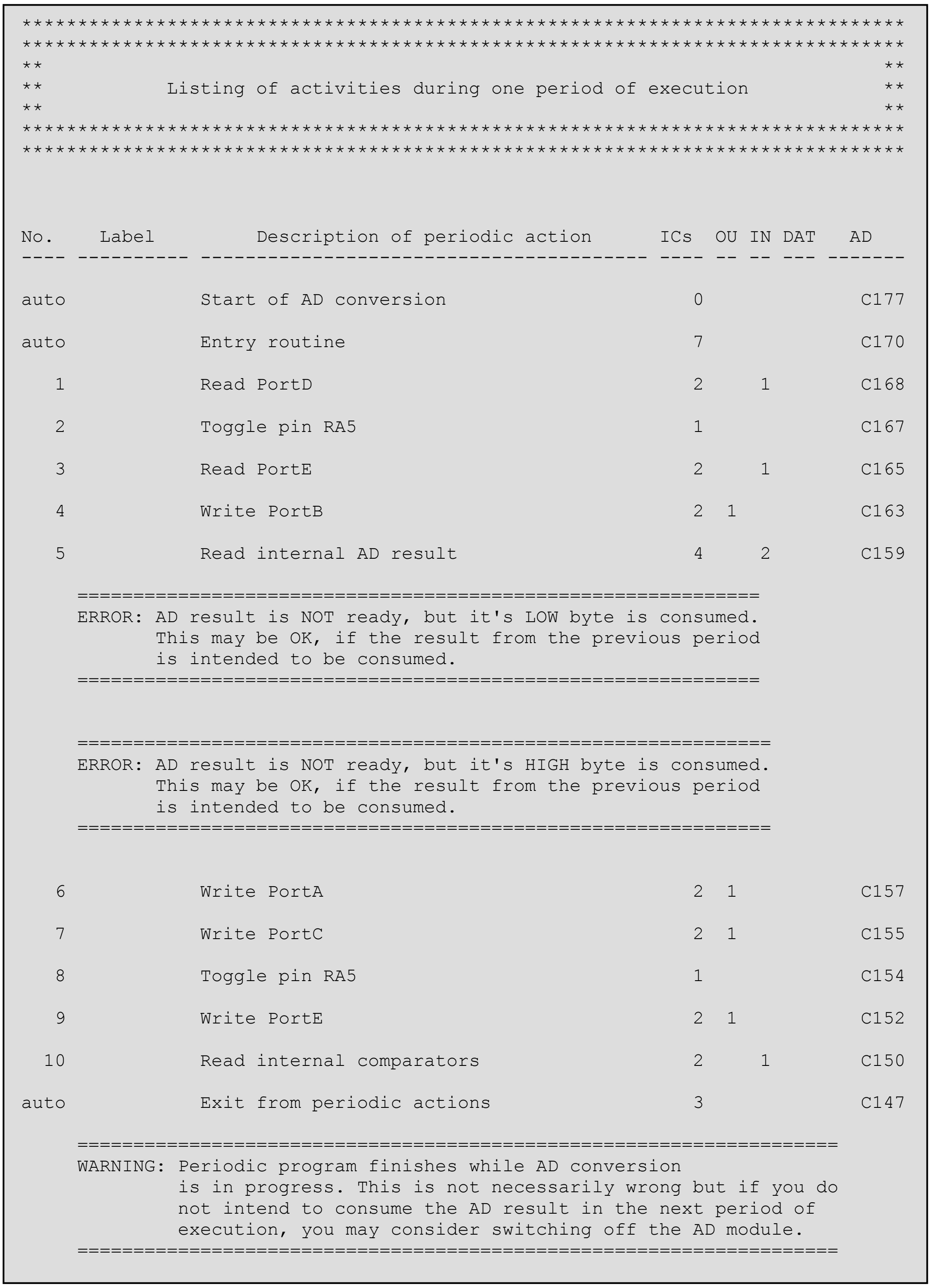
**Figure 72: The analysis report of the sample dummy program (try I, part I).**

This rather boring part of the report is shown here because of the legend, which we are going to utilize a bit more intensively than we did when discussing the periodic Hello world program. Also, we should pay our attention to the last two lines, where the analyzer noticed that the AD module is enabled. The last line informs us about the time requirements of the module. As we can see, the channel acquisition time (section 15.2.1, appendix B) is zero, as determined by line 5 of our listing. The AD conversion takes a whopping 177 PIC's instruction cycles to complete and deliver the result. Further, before another AD conversion starts, we need to wait additional 32 PIC's instruction cycles, or the next AD result may be inaccurate.

The second part of the report is presented in the Figure 73. The first activity is the start of AD conversion, which is taken care of automatically by PIC's hardware and therefore this step does not consume any instruction cycles (column `ICs`). In the last column (`AD`) of the report, we can follow the activity of the AD module. According to the legend in the Figure 72, the letter `C` followed by a three digit number (at most) denotes AD conversion in progress and the number of PIC's instruction cycles before the conversion is over. At the very beginning of the period there are all 177 cycles left before the result arrives.

The next activity is the automatically provided `Entry routine`, which is needed for proper working of our periodic program. This step takes 7 ICs to execute, so at the end of it there are *only* 170 ICs before the AD result is ready.

Here comes an important observation. The column AD shows the state of AD module **after** the listed activity has completed. This makes sense, since when certain line shows that AD result is ready then the very next line may consume the result immediately.



```
********************************************************************************
********************************************************************************
**                                                                            **
**           Listing of activities during one period of execution             **
**                                                                            **
********************************************************************************
********************************************************************************



No.    Label         Description of periodic action       ICs  OU IN DAT   AD
---- ---------- ----------------------------------------- ---- -- -- --- -------

auto            Start of AD conversion                       0              C177

auto            Entry routine                                7              C170

   1            Read PortD                                   2     1        C168

   2            Toggle pin RA5                               1              C167

   3            Read PortE                                   2     1        C165

   4            Write PortB                                  2  1           C163

   5            Read internal AD result                      4     2        C159

     ==============================================================
     ERROR: AD result is NOT ready, but it's LOW byte is consumed.
            This may be OK, if the result from the previous period
             is intended to be consumed.
     ==============================================================


     ===============================================================
     ERROR: AD result is NOT ready, but it's HIGH byte is consumed.
            This may be OK, if the result from the previous period
             is intended to be consumed.
     ===============================================================


   6            Write PortA                                  2  1           C157

   7            Write PortC                                  2  1           C155

   8            Toggle pin RA5                               1              C154

   9            Write PortE                                  2  1           C152

  10            Read internal comparators                    2     1        C150

auto            Exit from periodic actions                   3              C147

     =====================================================================
     WARNING: Periodic program finishes while AD conversion
              is in progress. This is not necessarily wrong but if you do
              not intend to consume the AD result in the next period of
              execution, you may consider switching off the AD module.
     =====================================================================
```

**Figure 73: The analysis report of the sample dummy program (try I, part II).**

The next line lists the first activity of our choice: the reading of port D. This step takes two instruction cycles (column `ICs`) and it writes one byte (the current state of the port D, of course) to the IN buffer (pipe), as the column `IN` reveals. After this step 168 ICs remain before the AD result is ready.

The second activity of ours is the already familiar toggling of pin (RA5), which takes one IC. Nothing else happens in this step, except that we are one IC closer to the AD result.

The third activity of ours reads port E and behaves in the same way as the activity number 1 does.

The fourth activity consumes one byte of OUT packet (pipe), as the column `OU` reveals, and writes it to the port B. Similarly, to the reading of the port, writing to it takes two ICs to execute.

Aha, the activity number five requires us to take a deep breath since the analyzer reports an error that is associated with it. Namely, activity 5 consumes the AD result, which is not ready yet. By examining the previous line, we deduce that we should wait for additional 163 instruction cycles before we can take a look at the result.

The discovered issue is reported as an `Error` but not as a `Fatal` error, since this kind of error does not lead to device crash or other erratic behaviour of eProDas system. Put it simply, if AD result is not ready, the command will read some gibberish instead of a meaningful value. Contrary, when the discovered issue may lead to device crash, deadlock or other erratic behaviour, the analyzer reports it as a `Fatal` error and it prevents further steps toward running of the periodic program.

Note also that the error is in fact reported as two separate errors, which is due to the fact that AD result is read in two chunks, since it occupies two bytes. The analyzer tries to be as precise as feasible when making the analysis and for that matter it pays attention to each part of the result separately.

Although errors were discovered, the analyzer continues with the analysis and tries to finish the report, so that it may list further errors to give us a chance to repair many mistakes in one development cycle.

Activities from 6 to 9 are nothing new and there is not much to be said about them. The reading of comparators' state (activity 10) is new with this regard, but still not special in any way; similarly to reading of some port this activity takes two instruction cycles and writes one byte to the IN packet. The last activity, the exit routine, is automatically provided by eProDas to take care of proper completion of the period.

Now we have to face a warning. The AD conversion, which was started as the first activity, is still in progress (it will complete 147 ICs after the period finishes). The system warns us that this AD conversion actually has no point, since its result is not consumed anywhere within the period. However, as the text hints, it may be possible to consume the result in the next period, if that was our intention. Therefore, the system cannot know for sure whether our program is wrong or right with this regard, so it does not report an error but a mere warning.

Before taking further steps, let us correct the reported issues. We must add the line that waits for completion of the AD conversion just before reading the AD result (line 16 in the Figure 74). Line 16 makes eProDas monitor AD hardware flag in a tight loop until the end of AD conversion is signalled by PIC's hardware. The accompanying report of the corrected program is presented in the Figure 75.

```
Status = eProDas_AddPeriodicAction_ReadPort(DeviceIdx, eProDas_Index_PORTD);
Status = eProDas_AddPeriodicAction_TogglePin(DeviceIdx, eProDas_Index_PORTA,5);
Status = eProDas_AddPeriodicAction_ReadPort(DeviceIdx, eProDas_Index_PORTE);
Status = eProDas_AddPeriodicAction_WritePort(DeviceIdx, eProDas_Index_PORTB);

Status = eProDas_AddPeriodicAction_WaitInternalAD(DeviceIdx); ← added line

Status = eProDas_AddPeriodicAction_ReadInternalAD(DeviceIdx);
Status = eProDas_AddPeriodicAction_WritePort(DeviceIdx, eProDas_Index_PORTA);
Status = eProDas_AddPeriodicAction_WritePort(DeviceIdx, eProDas_Index_PORTC);
Status = eProDas_AddPeriodicAction_TogglePin(DeviceIdx, eProDas_Index_PORTA,5);
Status = eProDas_AddPeriodicAction_WritePort(DeviceIdx, eProDas_Index_PORTE);
Status = eProDas_AddPeriodicAction_ReadInternalComparators(DeviceIdx);

Status = eProDas_AnalyzePeriodicActions(DeviceIdx, 1, 0, 1);
```

**Figure 74: Code for running the sample dummy periodic program (part I, try II).**

The picture is a bit more relaxing now since there are no signs of any errors and warnings in this part of the report. We may notice the activity number 5, which waits for completion of the AD result. According to the activity number 4, the waiting period must last for at least 163 ICs, but as we can see the line 5 takes 166 ICs to execute. This is due to the fact that the program loop, which monitors the state of AD module, needs some time to exit upon detection of the proper state.

```
********************************************************************************
********************************************************************************
**                                                                            **
**            Listing of activities during one period of execution            **
**                                                                            **
********************************************************************************
********************************************************************************



No.    Label          Description of periodic action       ICs  OU  IN  DAT   AD
---- ---------- ---------------------------------------- ---- -- -- --- -------

auto            Start of AD conversion                     0               C177

auto            Entry routine                              7               C170

   1            Read PortD                                 2      1        C168

   2            Toggle pin RA5                             1               C167

   3            Read PortE                                 2      1        C165

   4            Write PortB                                2   1           C163

   5            Wait for internal AD result              166           R1  G 29

   6            Read internal AD result                    4      2        G 25

   7            Write PortA                                2   1           G 23

   8            Write PortC                                2   1           G 21

   9            Toggle pin RA5                             1               G 20

  10            Write PortE                                2   1           G 18

  11            Read internal comparators                  2      1        G 16

auto            Exit from periodic actions                 3               G 13
```

**Figure 75: The analysis report of the sample dummy program (try II, part II).**

In fact, this analysis may be wrong for up to 2 ICs, since a true PIC simulator would be needed to correctly predict exactly when the state change is detected (or better: at which stage the monitoring loop is executing when the state changes). That is why the report stresses that it only estimates the execution time.

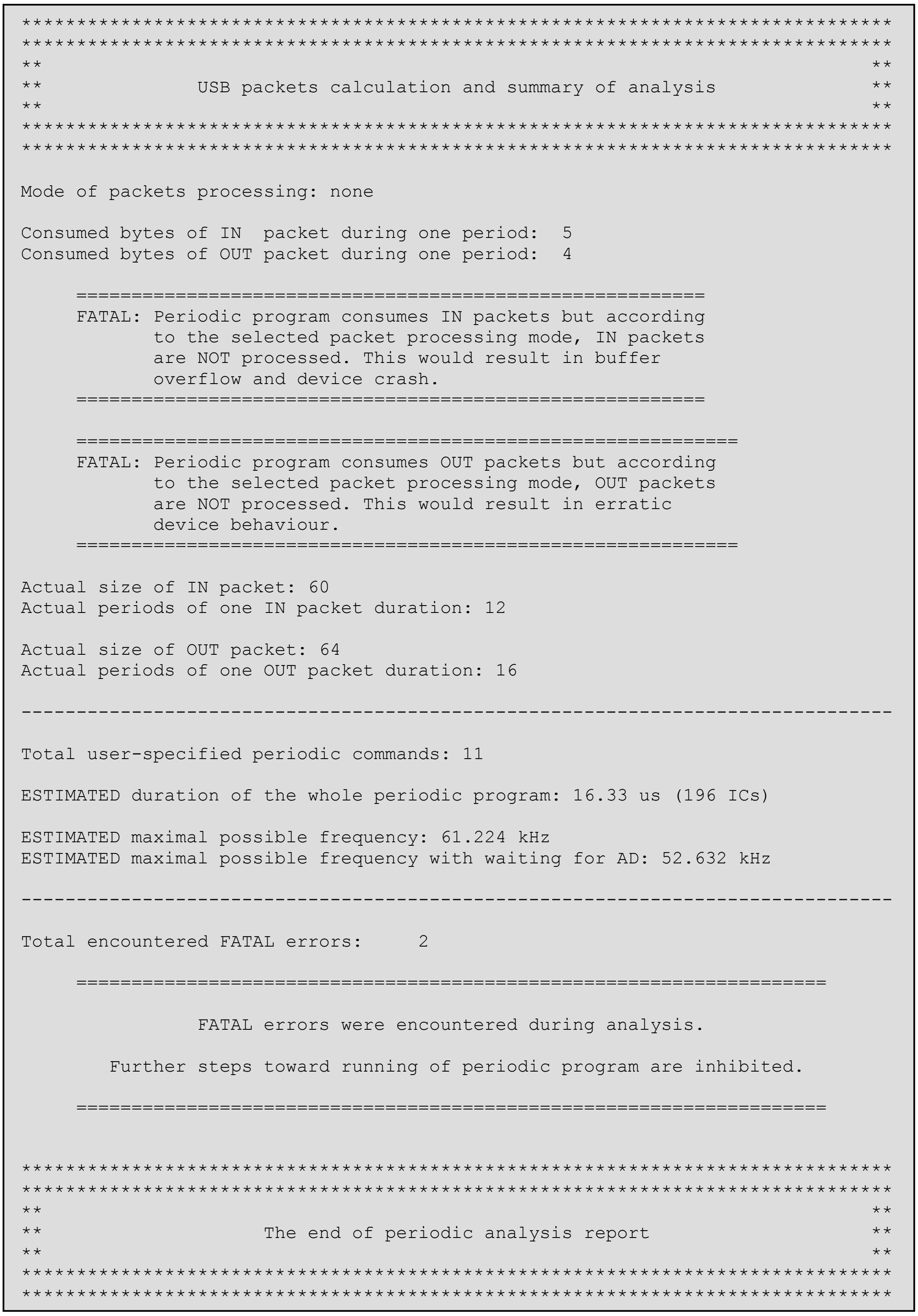
We also observe two new marks in the last column of the line with activity number 5. According to the legend (Figure 72) the letter `R` indicates that the AD result is ready. The number after the letter is the result count; this is the first AD result in this period of execution, hence the number 1.

There is also a `G` mark on the same line. According to the legend this signs that AD converter rests now and another AD conversion must not be started before this “gap” interval passes. Again, the number of PIC’s instructions until the end of the resting is signalled by the number after the `G`.

The Figure 76 presents the third part of the report. Huh, this one looks really depressing. The first informative line tells us that we selected no packets’ processing when configuring periodic actions. By taking a look at the line 11 in the Figure 71 we recall that this is indeed the case.

The next two lines reveal the calculated number of bytes that are written to IN pipe and read from OUT pipe, respectively, during one period of execution. Two self descriptive fatal errors follow immediately: IN as well as OUT buffers are being utilized and for that matter we ought to select the packets’ processing mode, which processes both directions of the transfer.

The related message near the end of the report informs as that due to the fatal errors our eProDas system does not let us proceed with further steps toward running the periodic program. It looks like we have no choice but to learn everything about the mysterious function `SetPeriodicMode`.



```
*******************************************************************************
*******************************************************************************
**                                                                           **
**              USB packets calculation and summary of analysis              **
**                                                                           **
*******************************************************************************
*******************************************************************************

Mode of packets processing: none

Consumed bytes of IN  packet during one period:  5
Consumed bytes of OUT packet during one period:  4

     =========================================================
     FATAL: Periodic program consumes IN packets but according
            to the selected packet processing mode, IN packets
            are NOT processed. This would result in buffer
            overflow and device crash.
     =========================================================

     ============================================================
     FATAL: Periodic program consumes OUT packets but according
            to the selected packet processing mode, OUT packets
            are NOT processed. This would result in erratic
            device behaviour.
     ============================================================

Actual size of IN packet: 60
Actual periods of one IN packet duration: 12

Actual size of OUT packet: 64
Actual periods of one OUT packet duration: 16

-------------------------------------------------------------------------------

Total user-specified periodic commands: 11

ESTIMATED duration of the whole periodic program: 16.33 us (196 ICs)

ESTIMATED maximal possible frequency: 61.224 kHz
ESTIMATED maximal possible frequency with waiting for AD: 52.632 kHz

-------------------------------------------------------------------------------

Total encountered FATAL errors:     2

     ====================================================================

                FATAL errors were encountered during analysis.

        Further steps toward running of periodic program are inhibited.

     ====================================================================


*******************************************************************************
*******************************************************************************
**                                                                           **
**                    The end of periodic analysis report                    **
**                                                                           **
*******************************************************************************
*******************************************************************************
```

**Figure 76: The analysis report of the sample dummy program (try II, part III).**

## 19.6 SetPeriodicMode (please, also read section 19.12.10)

Prototype in C/C++

```
int eProDas_SetPeriodicMode(unsigned int DeviceIdx, unsigned int Mode,
  unsigned int OUT_Periods, unsigned int IN_Periods,
  unsigned int Const_OUT_Length);
```

Prototype in Delphi

```
function eProDas_SetPeriodicMode(DeviceIdx: LongWord; Mode: LongWord;
  OUT_Periods: LongWord; IN_Periods: LongWord; Const_OUT_Length: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_SetPeriodicMode (ByVal DeviceIdx As UInteger,
  ByVal Mode As UInteger, ByVal OUT_Periods As UInteger,
  ByVal IN_Periods As UInteger, ByVal Const_OUT_Length As UInteger) As Integer
```

Parameter `Mode` selects mode of transactions handling whereas the remaining three parameters fine-tune the selection, if they are set to a non-zero value. The modes are presented in the following table.

| Mode | associated named constant | action at start | duration at start [ICs] | action at end | duration at end [ICs] |
|---|---|---|---|---|---|
| 0 | eProDas_PeriodicMode_None | | 0 | | 3 |
| 1 | eProDas_PeriodicMode_Out | OUT | 13 | | 3 |
| 2 | eProDas_PeriodicMode_InAtStart | IN | 13 | | 3 |
| 3 | eProDas_PeriodicMode_InAtEnd | | 0 | IN | 22 |
| 4 | eProDas_PeriodicMode_BothAtStart | OUT+IN | 29 | | 3 |
| 5 | eProDas_PeriodicMode_Synchronous | OUT+IN | 20 | | 3 |
| 6 | eProDas_PeriodicMode_Out_InAtEnd | OUT | 13 | IN | 22 |
| 7 | eProDas_PeriodicMode_OutConst12 | C.OUT | various | | 3 |
| 8 | eProDas_PeriodicMode_OutConst6_InAtStart | C.OUT+IN | various+13 | | 3 |
| 9 | eProDas_PeriodicMode_OutConst6_InAtEnd | C.OUT | various | IN | 22 |
| 10 | eProDas_PeriodicMode_OutConst8_InAtStart | C.OUT+IN | various+13 | | 3 |
| 11 | eProDas_PeriodicMode_OutConst8_InAtEnd | C.OUT | various | IN | 22 |
| 12 | eProDas_PeriodicMode_OutConst10_InAtStart | C.OUT+IN | various+13 | | 3 |
| 13 | eProDas_PeriodicMode_OutConst10_InAtEnd | C.OUT | various | IN | 22 |

**Table 10: Periodic modes and packets processing.**

The basic decision to be made is whether to process OUT and/or IN packets or not. If OUT packets are used, they are always processed at the beginning of the period (the left `USB transactions` rectangle in the Figure 70) whereas for IN packets we have the choice of processing them either at the beginning of the period (possibly together with OUT packets; the left `USB transactions` rectangle in the Figure 70) or at the end (the right `USB transactions` rectangle in the Figure 70). Further, OUT packets exist in two incarnations: true OUT packets or constant OUT buffer (`C.OUT`). The former are the OUT packets that the section 19.4 describes. The latter is nothing more than a constant circulating buffer, which periodic commands read in the same way as it were the true OUT packets.

If your application needs to supply different OUT data all the time (like when implementing a control system) during periodic actions, select utilization of true OUT packets. On the other hand, if the setup requires sending of OUT packets with repetitive contents and the buffering capabilities of the PIC suffice, you can preload the contents of OUT packets to the PIC's memory. In this case your application does not send any OUT packets during the running of the periodic program and the whole USB capacity is dedicated to the transfer of IN packets. For example, if you generate a sine wave by writing the proper pattern to some DA converter you need to indefinitely supply that pattern over and over again. If the whole pattern fits into device's constant OUT buffer, you do not need to bother with it during the running of periodic actions, but you merely preload it before the actions are started.

### 19.6.1 Description of various periodic modes

We already know about the mode `eProDas_PeriodicMode_None` (0), which selects no data exchange between the application and the device during periodic actions. Situations with such requirement are rare but they exist nonetheless. As an example, you may program your device to be a converter between SPI bus and parallel port. The device merely reads the data from SPI bus and writes it to the parallel port of your choice (say, port D). In this case no data needs to be exchanged between the application and the device. When selecting this mode the last three parameters of the function `SetPeriodicMode` must be set to zero.

To utilize true (i.e. non-constant) OUT packets, select one of the modes `PeriodicMode_Out` (1), `PeriodicMode_BothAtStart` (4), `PeriodicMode_Synchronous` (5), or `PeriodicMode_Out_InAtEnd` (6). The first choice selects processing of OUT packets only, so data travels from the host PC to the eProDas device but not in the opposite direction; typically, this is suitable when implementing some sort of plain signal generation. Other listed modes enable OUT as well as IN packets processing so data can travel into both directions, like signal generation + measurements.

When OUT packets are enabled eProDas executes a small piece of code at the beginning of each period of execution. This code checks whether the currently used OUT buffer is exhausted (periodic program has read all contained data) and if so the buffer is declared empty and ready for reception of new data from the host PC. The next OUT buffer becomes the current one. If this newly declared current buffer is not empty (it has already received data) all is well and done. However, if the buffer is empty, eProDas stops the normal flow of execution, increments internal counter of buffer *underruns* and then it waits in a tight program loop that the buffer is filled in with the data that the host PC will send somewhere in the future.

To utilize IN packets, select one of the modes `PeriodicMode_InAtStart` (2), `PeriodicMode_InAtEnd` (3), `PeriodicMode_BothAtStart` (4), `PeriodicMode_Synchronous` (5), `PeriodicMode_Out_InAtEnd` (6), or any of the modes from 8 to 13. The first two listed modes select processing of IN packets only, so data travels from the device to the host PC but not in the opposite direction (like measurements without any sort of signal generation). Modes from 4 to 6 enable both OUT and IN packets and consequently bidirectional flow of data. Modes from 8 to 13 enable utilization of constant OUT buffer together with IN packets.

When processing IN packets you can instruct eProDas to process these before the user's periodic program of after it. Any mode with literal `InAtStart` in the name of the associated constant as well as `PeriodicMode_BothAtStart` (4) and `PeriodicMode_Synchronous` (5) all process IN packets before the execution of the periodic program, whereas modes with literal `InAtEnd` move processing of IN packets to the end of the period.

Whenever IN packets are processed a small piece of code is executed either before or after the user's periodic program. This code checks whether the currently used IN buffer is full (periodic program has written the prescribed amount of data to it) and if so the buffer is declared ready for transmission to the host PC. The next IN buffer becomes the current one. If this newly declared current buffer is empty (it has already sent the data to the host PC) all is well and done. However, if the buffer is still full, eProDas stops the normal flow of execution, increments internal counter of buffer *overruns* and then it waits in a tight program loop for the buffer to get empty by sending its data to the host PC somewhere in the future.

Note the difference between modes `BothAtStart` and `Synchronous`. Both modes process OUT as well as IN packets at the beginning of the period. However, in the former case OUT and IN packets are processed independently, whereas by the latter selection both buffers are swapped synchronously, which saves some time (9 ICs) but it demands that both types of packets experience equal duration.

### 19.6.2 Size of USB packets

eProDas periodic engine requires that all OUT packets are of the same size during periodic experiment. Also all IN packets must be equally long. The size of OUT packets may be different than the size of IN ones, though. Further, since packets are processed only before or after the periodic program is executed, the size of OUT (IN) packets must be an exact multiple of the number of bytes that your periodic program reads from (writes to) the OUT (IN) pipe in one period. An additional limitation is imposed by the USB bus, which dictates the maximal packets' sizes of 64 bytes (if you are familiar with USB standard: eProDas uses bulk USB transfer).

eProDas needs to know packets' sizes before periodic program starts so that it can tune its periodic engine accordingly (the tuning happens when the function `PreparePeriodicActions` is executed). You can let eProDas calculate packets' sizes by itself or you can influence the calculations. To let eProDas do the calculations for OUT and/or IN packets by itself, set the respective parameter `OUT_Periods` and/or `IN_Periods` to zero. In this case eProDas selects as large packets as feasible in order to minimize the USB protocol overhead. By setting one or both of these parameters to a non-zero value, you determine for how many periods of execution the respective packets last.

**Note.** When `Synchronous` mode of operation is selected, you must set both `OUT_Periods` and `IN_Periods` to the same value. Alternatively, set only one of these parameters to a non-zero value and eProDas automatically assumes that the demand is also valid for the other type of packets.

As an example, let us take a look at the report in the Figure 76 one more time. By examining the first few lines of the text we recall that the sample dummy periodic program writes 5 bytes into the IN pipe during each period and that it also reads 4 bytes out of the OUT pipe.

```
Consumed bytes of IN  packet during one period:  5
Consumed bytes of OUT packet during one period:  4
```

After the two fatal error messages we can discover the following text.

```
Actual size of IN packet: 60
Actual periods of one IN packet duration: 12

Actual size of OUT packet: 64
Actual periods of one OUT packet duration: 16
```

Despite the errors, eProDas tries to calculate optimal packets' sizes and it proposes that the size of IN packets is 60 bytes; this is the maximal number less than or equal to the hard limit of 64 bytes that is divisible by 5. In this scenario one IN buffer is filled in with data and sent to your application once per 12 periods of execution. Since IN packets are 60 bytes long, we are unable to utilize the whole IN buffers; 4 bytes of each IN buffer are completely unexploited but this is nothing terribly tragic.

The proposed size of OUT packets is 64 bytes so one OUT buffer is consumed once per 16 periods of execution and consequently that is the average speed at which your application must send new OUT packets to the device during the experiment.

Now, let us finally make this sample dummy example correct and its report error free by replacing the line 11 in the Figure 71 with the following content.

```
11. Status = eProDas_SetPeriodicMode(DeviceIdx,
                                      eProDas_PeriodicMode_Out_InAtEnd, 4,3,0);
```

**Figure 77: Correction of the sample dummy periodic program (part I, try III).**

This time we should be able to pass the analysis, as we selected correct mode of operation where both types of packets are processed. Although not needed to make the example work, we also influenced the calculation of packets' sizes by instructing that OUT packets last 4 periods of execution, whereas the duration of IN packets is 3 periods. We can examine the resulting report in the Figure 78.

```
*******************************************************************************
*******************************************************************************
**                                                                           **
**           Listing of activities during one period of execution           **
**                                                                           **
*******************************************************************************
*******************************************************************************

No.    Label         Description of periodic action      ICs  OU IN DAT   AD
---- ---------- ---------------------------------------- ---- -- -- --- -------

auto            Start of AD conversion                      0              C177

auto            Entry routine                               7              C170

auto            Formation of USB OUT packets               13              C157

   1            Read PortD                                  2     1        C155

   2            Toggle pin RC7                              1              C154

   3            Read PortE                                  2     1        C152

   4            Write PortB                                 2  1           C150

   5            Wait for internal AD result               153           R1 G 29

   6            Read internal AD result                     4     2        G 25

   7            Write PortA                                 2  1           G 23

   8            Write PortC                                 2  1           G 21

   9            Toggle pin RA2                              1              G 20

  10            Write PortE                                 2  1           G 18

  11            Read internal comparators                   2     1        G 16

auto            Processing of IN packets & exit            22              idle
```

**Figure 78: The analysis report of the sample dummy program (try III, part II).**

This report differs a bit from the one in the Figure 75. Now there exists the third `auto` entry named `Formation of USB OUT packets`, which takes 13 ICs to execute. The waiting for the AD result (the fifth activity) has been shortened from 166 ICs to 153 ICs, which is a consequence of more work (OUT packets processing) being done before the waiting phase begins. Also, the last auto activity changed from plain `Exit` to `Processing of IN packets & exit`; it takes 22 ICs to execute. As the last observation, note that now AD conversion gap ends before the end of the period, so upon exit the AD converter is `idle` and ready to start new AD conversion immediately.

The summary of the analysis is presented in the Figure 79. The `Mode of packets processing` reflect our choice in the Figure 77 and packets of both types are now processed as it is required for bidirectional flow of data. Within the report we can spot the following two new lines, which reveal that we influenced packets calculation instead of letting eProDas determine optimal packets' sizes.

```
Requested periods of one IN  packet duration:  3
Requested periods of one OUT packet duration:  4
```

```
*******************************************************************************
*******************************************************************************
**                                                                           **
**              USB packets calculation and summary of analysis              **
**                                                                           **
*******************************************************************************
*******************************************************************************

Mode of packets processing: OUT at the beginning, IN at the end

Consumed bytes of IN  packet during one period:  5
Consumed bytes of OUT packet during one period:  4

Requested periods of one IN  packet duration:  3
Requested periods of one OUT packet duration:  4

Actual size of IN packet: 15
Actual periods of one IN packet duration:  3

Actual size of OUT packet: 16
Actual periods of one OUT packet duration:  4

-------------------------------------------------------------------------------

Total user-specified periodic commands: 11

ESTIMATED duration of the whole periodic program: 17.92 us (215 ICs)

ESTIMATED maximal possible frequency: 55.814 kHz

NOTE: Calculated maximal frequencies do NOT
      take into account potential USB bottlenecks.

*******************************************************************************
*******************************************************************************
**                                                                           **
**                    The end of periodic analysis report                    **
**                                                                           **
*******************************************************************************
*******************************************************************************
```

**Figure 79: The analysis report of the sample dummy program (try III, part III).**

Accordingly, the size of IN packets is now 15 bytes so that a new packet can be sent to the host PC once per three periods of execution. Similarly, OUT packets are now 16 bytes long so that the device accepts new packet from the host PC once per four periods of execution.

No fatal errors have been discovered during the analysis so this program can be run. The maximal frequency of execution is estimated to be slightly less than 56 kRuns/s.

#### 19.6.2.1 When to manually decrease the duration of USB packets

It seems a bit odd to have a choice of decreasing the default duration of USB packets (by setting the parameters `OUT_Periods` and/or `IN_Periods` accordingly), as by doing it we increase the USB protocol overhead and consequently decrease data throughput. However, in certain scenarios of usage, other properties are more important than the large amount of transferred data.

Decreasing of USB packets is useful in the cases of running periodic actions at low frequencies. Imagine that you want to measure temperature of a lake once per hour. Your periodic program consists of a single execution step, which is reading of AD result that represents output of a temperature sensor. As we already know, the result of AD conversion occupies two bytes of IN packet. Without our intervention eProDas aims to exploit USB bandwidth as efficiently as possible, which means that it would stretch IN packets to be 64 bytes long. Consequently, the data would be sent to the host PC once per 32 periods of execution when data amount of 64 bytes is produced.

This scheme works well for high frequency experiments but not for our particular case. According to the proposed scheme you need to wait 32 hours to obtain the first 32 results, after the next 32 hours you receive the next bunch of them and so on. You certainly do not have the patience to wait that long and you would like to receive each result of the measurement as soon as it is available. For that matter eProDas possesses the possibility of decreasing USB packets' sizes. You can demand that each result of temperature measurement is delivered to you immediately when taken, simply by setting the parameter `IN_Periods` to 1.

Decreasing of OUT packets may be useful too. Imagine that you are not implementing ordinary signal generator but some sort of control system. These systems (regulators) adjust their output according to the measurement of some inputs (controlled variable). In such cases you do not want to influence the output only after, say, 32 periods of execution, when the current OUT packet expires, but you want to change the output value as instantly as possible. Delays (lags) are extremely bad for control systems, since they decrease phase margins and destabilize the whole system.

**Note.** Without your intervention eProDas always try to exploit USB bandwidth as efficiently as possible. By altering the packets' sizes you can only decrease the peak possible data throughput but you cannot improve it. Therefore, do not alter packets' sizes if the reason is other than improving **responsiveness** of **slowly** running periodic actions.

### 19.6.3 To process IN packets at the beginning or at the end, that is the question

Let us do one more adjustment to the sample dummy program in the Figure 71 (with the correction in the Figure 77). Instead of selecting mode `PeriodicMode_Out_InAtEnd`, we are going to experiment with the choice `PeriodicMode_BothAtStart`, according to the following figure.

```
Status = eProDas_SetPeriodicMode(DeviceIdx,
                                     eProDas_PeriodicMode_BothAtStart, 4,3,0);
```

**Figure 80: Correction of the sample dummy periodic program (part I, try IV).**

Now OUT as well as IN packets are processed at the beginning of the execution period, as the third `auto` action in the Figure 81 reveals. Note that waiting for AD converter has been further decreased from 153 ICs (Figure 78) to 137 ICs, which is again a result of more work done (processing of both types of packets instead of just OUT ones) before the waiting starts.

Consequently there is less work to do after AD conversion finishes, so the periodic program can stop working sooner. As we can see near the end of the Figure 81, it is now possible to run the program at a frequency of 61 kHz. However, doing so would produce inaccurate AD results in the following periods, since AD converter would be forced to start at the beginning of the next period before its latency time is over. To respect AD's dynamic, we should not run this program above the frequency of 57 kHz or so (additional 13 ICs that are needed for the AD converter to enter the `idle` state; see the last `auto` activity, column AD), which the report also informs us about.

The gain may not be radical in comparison to the previous example (57.4 kHz instead of 55.8 kHz), but if some other work was done after AD converter finishes, we could gain up to 19 ICs by moving the processing of IN packets to the beginning of the period.

#### 19.6.3.1 So, why do not we always process IN packets at the beginning?

Yes, indeed. Why do we have this pesky choice of processing IN packets either at the beginning or at the end? Obviously, IN packets at the beginning accelerate programs or they at least do not slow them down? As you might suspect there is a catch and you need to be aware of it before you hurry with packets formation to the front of the train.

```
No.    Label         Description of periodic action      ICs  OU IN DAT    AD
---- ---------- ---------------------------------------- ---- -- -- --- -------

auto            Start of AD conversion                      0              C177

auto            Entry routine                               7              C170

auto            Formation of USB OUT & IN packets          29              C141

   1            Read PortD                                  2     1        C139

   2            Toggle pin RC7                              1              C138

   3            Read PortE                                  2     1        C136

   4            Write PortB                                 2  1           C134

   5            Wait for internal AD result               137           R1 G 29

   6            Read internal AD result                     4     2        G 25

   7            Write PortA                                 2  1           G 23

   8            Write PortC                                 2  1           G 21

   9            Toggle pin RA2                              1              G 20

  10            Write PortE                                 2  1           G 18

  11            Read internal comparators                   2     1        G 16

auto            Exit from periodic actions                  3              G 13

*******************************************************************************
*******************************************************************************
**                                                                           **
**              USB packets calculation and summary of analysis              **
**                                                                           **
*******************************************************************************
*******************************************************************************

Mode of packets processing: OUT & IN at the beginning

             . . . some of the repeated contents are dropped . . .

Total user-specified periodic commands: 11

ESTIMATED duration of the whole periodic program: 16.33 us (196 ICs)

ESTIMATED maximal possible frequency: 61.224 kHz
ESTIMATED maximal possible frequency with waiting for AD: 57.416 kHz

NOTE: Calculated maximal frequencies do NOT
      take into account potential USB bottlenecks.
```

**Figure 81: The analysis report of the sample dummy program (try IV, part II & III).**

Let us examine the execution of the last version of our sample dummy program with the help of the Figure 69 (page 187). The first period starts at this very moment (AD conversion starts...) then there comes the processing of IN packets.

The code checks the so far written data to the current IN buffer, which is zero, since we have not done anything yet. The buffer contains less than the prescribed amount of 15 bytes (the amount that is generated in three periods), so nothing happens with the IN buffer.

Next, our program is executed for the first time and five bytes are written to the current IN buffer within the process.

At the beginning of the next period the code for processing IN packets discovers that the current size of IN buffer is 5 bytes, which is less than 15 bytes, so nothing happens. Then our program is executed for the second time and additional 5 bytes are written to the IN buffer.

At the *beginning* of the third period the size of IN buffer is 10 bytes, so nothing happens. Now, the program is executed for the third time and at the *end* of the execution the size of IN buffer is 15 bytes. The IN packet is full and it is in the proper form to be sent to the host PC. However, nothing checks this state now, since IN packets are not processed at the end of the period. The IN packet is sleeping until the next period starts and the code for processing of IN packets discovers that the IN buffer is ready for transmission to the host PC.

Nothing is inherently wrong with such sequence of events and *average* data throughput is not lowered by such arrangement. The only difference between processing of IN packets at the beginning or at the end is that *transmission* of IN packets is shifted in time by one period.

Now, imagine that you do one AD conversion per hour and that you also demanded IN packets to be delivered after each period (`IN_Periods` = 1). The first time when periodic program is executed the result will be written to the IN packet and it will remain to sit there until one hour passes and eProDas engine discovers in the next period that the packet is ready for transmission. You will constantly receive the AD result one hour late although it is correctly sampled on time and stored internally on the chip.

The bottom line is, when you intend to execute periodic programs at high data rates so that generally you cannot percept each result separately and you can tolerate delayed of-one-period-later *delivery* of the results (which are still sampled at the *correct* instants in time), instruct IN packets formation at the beginning of your program to gain the speed. For slowly executing periodic actions you generally care more about on-time delivery than the speed of execution so processing of IN packets at the end works better for you.

### 19.6.4 Using constant OUT buffer instead of true OUT packets

Constant OUT buffer is more comfortable to work with than sending of OUT packets to the device, so you are welcome to use this feature whenever your setup permits it. When constant OUT buffer is operational eProDas transforms the six OUT buffers with capacity of 64 bytes into one contiguous OUT buffer memory. Your periodic program then reads this buffer in the same way as it would do the true OUT buffers. When all bytes from constant OUT buffer are read the internal pointer is automatically reinitialized to the beginning of the buffer and the periodic program starts to read the same values over and over again.

The parameter `Const_OUT_Length` of the function `SetPeriodicMode` must be set to the length of the buffer. Since the buffer is a contiguous chunk of memory, a 64-byte boundary may be crossed during one period of execution without problems. However, since the buffer is still processed (i.e. checked for its pointer reinitialization) only at the beginning of the period, its length must be an exact multiple of one period’s consumption.

For example, if your periodic program consumes 17 bytes of OUT data during each period, the permitted values of the parameter `Const_OUT_Length` are 17, 34, 51, 68, 85… up to the maximal length that does not cross the capacity of the entire buffer. Note the listed option 68, which stresses that 64-byte boundary does not pay any role whatsoever.

Upon calling the function `SetPeriodicMode` eProDas does not know your periodic program yet, so it cannot check, whether you specified a proper buffer length in accordance with the above rule. However, when the analysis of the program is done, a fatal error will be claimed if you try to cheat.

Let us now take a look at the Table 10 (page 197) again, where we can discover a myriad of possible modes, which deal with the constant OUT buffer. What is going on here?

The two modes `OutConst6_InAtStart` and `OutConst6_InAtEnd` are the closest analogies to the already known modes `BothAtStart` and `Out_InAtEnd`, respectively. The number 6 in the name of the constant reminds as that the six OUT buffers are utilized as the constant OUT buffer. Consequently, in these two modes the maximal length of the buffer is 384 bytes (6 x 64 bytes).

How about if you need more space and to get it you are willing to live with less than six IN buffers although this means lower data throughput? For that matter the two modes `OutConst8_InAtStart` and `OutConst8_InAtEnd` devote two IN buffers in addition to the six OUT buffers for the task of providing larger constant OUT buffer, which has now the capacity of 512 bytes (8 x 64 bytes). As far as your periodic program and the application on a host PC are concerned, there is no difference in operation. Not a single line of code needs to be adjusted or changed for this to work. The only difference is that due to a lower count of IN buffers (four instead of six) you may not be able to successfully (i.e. without buffer overruns) run your application at as high frequencies as before.

To increase the constant OUT buffer even further, use one of the two modes `OutConst10_InAtStart` and `OutConst10_InAtEnd`. Now, there are only two IN buffers left, but the maximal capacity of the constant OUT buffer is 640 byte (10 x 64 bytes).

Finally, if you do not need IN packets at all, you can devote all 6 IN buffers to the constant OUT buffer by selecting the mode `OutConst12`, which is analogous to the plain `Out` mode. The maximal capacity of the resulting constant OUT buffer is now 768 bytes (12 x 64 bytes).

### 19.6.4.1 Overhead of constant OUT buffer processing

By examining the Table 10 (page 197) one more time, we can see that the fourth and the sixth columns specify the time overhead (durations) that certain mode of operation imposes on the program execution. However, when dealing with constant OUT buffer, the table gives the word `various` instead of meaningful specifications. The reason is that the processing time depends on the length of the buffer (i.e. the value of the parameter `Const_OUT_Length`) according to the following table.

| row | buffer length [bytes] | duration [ICs] |
|---|---|---|
| 1 | general | 8 |
| 2 | general less than 512 | 7 |
| 3 | general less than 256 | 4 |
| 4 | 2, 4, 8, 16, 32, 64, 128, 256 | 2 |
| 5 | 255 | 3 |
| 6 | 257, 258, 260, 264, 272, 288, 320, 384 | 5 |
| 7 | 513, 514, 516, 520, 528, 544, 576, 640 | 6 |
| 8 | 512 | 3 |
| 9 | 511 | 5 |

**Table 11: Overhead of constant OUT buffer processing.**

In general case it takes 8 PIC's instructions (the first row) to check whether the 16-bit (12-bit in fact) pointer reached the end of the buffer and to reinitialize it to the beginning of the buffer. When the length of the buffer is less than 256 bytes, we can only check and reinitialize the lower byte of the pointer, so the needed time drops to 4 ICs (the third row). Although it seems a bit odd it is possible to optimize pointer checking even in the general case if the buffer length is less than 512 bytes (7 ICs, the second row).

Further optimizations are possible, as the additional rows of the table reveal. If the buffer length is an integer power of 2 (like $2^1$ , $2^2$…) and less than or equal to 256, it can be handled by burning only 2 ICs (the fourth row of the table). This is an extremely attractive option. Buffers of lengths in the form of an exact power of 2 are important in data-acquisition world, since these are the lengths that allow you to calculate a Fast-Fourier-Transform (FFT) on the buffer.

The length of 255 can also be optimized so that it burns 3 ICs instead of 4 ones (the fifth row). Such option is also attractive since 255 is a period of a pseudo-random-binary-signal (PRBS) that is generated with 8-bit shift register (8-bit registers present some difficulties when generating PRBS in hardware, but when calculating PRBS values purely in software this peculiarity does not matter that much). PRBS is an extremely important excitation signal in many data-acquisition setups.

A bit less efficient (but still better than the general case) is the processing of lengths in the form $256+2^n$ (the sixth row). You cannot run FFT on buffers of these lengths but you may exploit them for other purposes. The same conclusion goes for the lengths in the form $512+2^n$ (the seventh row).

The last two rows of the table again present two attractive options. To utilize buffer of the length of 512 bytes you only need to burn 3 ICs. This is the maximal possible length of the constant OUT buffer that allows you to calculate FFT on it.

The buffer length of 511 is again important because this is the period of PRBS that is generated with 9-bit shift register. And this is the maximal possible length of the constant OUT buffer that allows you to implement PRBS excitation without true OUT packets.

## 19.7 Clock for periodic actions

Clock for determining frequency (period) of periodic actions is synthesized by dividing 12 MHz PIC's instruction clock by a user specified 16-bit integer divider. The division is done by PIC's hardware for which the same timer that is in charge of PWM2 duty cycle (section 15.5.4) is exploited. Consequently, module PWM2 cannot be active during periodic actions.

**Example.** Divider of 120 leads to periodic actions that run at frequency of 12 MHz / 120 = 100 kHz. When divider is increased to 1,200, the frequency of periodic actions drops to 10 kHz. Some dividers that lead to round frequencies are listed in the Table 12.

| **divider** | **resulting frequency [Hz]** |
|---:|---:|
| 48 | 250,000 |
| 60 | 200,000 |
| 120 | 100,000 |
| 240 | 50,000 |
| 600 | 20,000 |
| 1,200 | 10,000 |
| 3,000 | 4,000 |
| 12,000 | 1,000 |
| 60,000 | 200 |

**Table 12: Some dividers for obtaining round frequencies of periodic actions.**

Any divider between 1 and 65,536 ($2^{16}$) is permissible so by making the selection of 1,789 we get periodic actions at frequency of 12 MHz / 1,789 = 6,707.658 Hz. The lowest possible periodic frequency according to this scheme is 12 MHz / 65,536 ≈ 183 Hz, which is too high for many natural science experiments. For example, when measuring temperature of a lake or biomass of some plant periods of minutes, hours or even days are preferred.

One way to achieve lower frequencies of execution is to let the system run at e.g. 200 Hz and perform the decimation of results on a host PC side. The concept works but the drawback is that PC is constantly occupied by reading results, most of which are discarded.

In order to lower periodic frequency in a more aesthetical way, an optional 24-bit *extended* divider is provided, which is implemented in software. When extended divider is in use, the actual periodic clock is obtained as follows. First, 12 MHz clock is divided by the previously described 16-bit divider, which is now called the *main* divider. In the second step the resulting intermediate clock is further divided by the extended divider.

**Example.** The maximal main and extended dividers are 65,536 ($2^{16}$) and 16,777,216 ($2^{24}$), respectively. The lowest possible periodic clock equals 12 MHz / 65,536 / 16,777,216 = $1.09 \cdot 10^{-5}$ Hz or the period of about 25 hours.

Therefore, it is possible to make eProDas system execute one periodic action per day. To obtain one period in about 86,400 seconds that constitute one nominal day you select e.g. 64,800 for main divider and 16,000,000 for extended divider. If you happen to be an astronomer, who lives in the belief that the length of the day is 23 hours + 56 minutes + 4.09054 seconds you would instead select main divider of 62,480 and extended divider of 16,548,801. Of course, you do not have to guess these values. Instead, eProDas API functions help you select them.

**Note 1.** The ultimate clock accuracy is confined to relative uncertainty of about $10^{-5}$ to $10^{-4}$ by the already mentioned imperfections of quartz crystal and oscillator circuitry (see the example in the section 19.3.12). This uncertainty approximately equals the error of 1 to 10 seconds per day. Although, eProDas API lets you specify nominal period of periodic actions to be, say, 22 hours + 7 µs, do not expect to achieve this level of accuracy unless you are a lucky guy who drive PIC's clock by temperature compensated quartz, oven controlled quartz or even better (but enormously expensive) atomic clock. If you resort to these high(er) precision options do not forget that you must drive the PIC's OSC1 pin with 48 MHz clock and not 4 MHz since internal phase locked loop (PLL) limits clock stability to ±0.25 % according to the datasheet.

**Note 2.** If you intend to use low frequencies that need extended divider for their synthesis we recommend you to read appendix C with some detailed explanations about it.

## 19.7.1 Setting the periodic clock

We have already learned about the two functions `SetPeriodicClockFrequency` (section 19.3.8.1) and `SetPeriodicClockPeriod` (section 19.3.8.2) that enable us to set the clock in a comfortable way. Internally, both of these functions rely on the function `SetPeriodicClockRaw` to do their job.

### 19.7.1.1 SetPeriodicClockRaw

Prototype in C/C++

```
int eProDas_SetPeriodicClockRaw(unsigned int DeviceIdx, unsigned int MainDivider,
  unsigned int ExtendedDivider, unsigned int Forced);
```

Prototype in Delphi

```
function eProDas_SetPeriodicClockRaw(DeviceIdx: LongWord; MainDivider: LongWord;
  ExtendedDivider: LongWord; Forced: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_SetPeriodicClockRaw (ByVal DeviceIdx As UInteger,
  ByVal MainDivider As UInteger, ByVal ExtendedDivider As UInteger,
  ByVal Forced As UInteger) As Integer
```

The word “Raw” in the name of the function reveals that frequency of periodic clock is selected by specifying raw parameters (dividers) of clock generating subsystem. The names of parameters are self explanatory. `MainDivider` specifies divider that divides 12 MHz clock in hardware. For lower frequencies than about 200 Hz, additional extended divider is specified by parameter `ExtendedDivider`. To disable extended divider set parameter `ExtendedDivider` to zero.

When parameter `Forced` is set to zero, the specified `MainDivider` must be greater than or equal to the minimal permissive main divider that the analysis of periodic actions calculates or else an error `eProDas_Error_ClockTooHigh` will result. By setting the parameter `Forced` to a non-zero value, eProDas is forced to accept any technically feasible `MainDivider`; when main divider is not smaller than or equal to 65,536 then an error `eProDas_Error_BadMainDivider` is triggered.

`ExtendedDivider` must not be greater than 16,777,216 in order to avoid triggering the error `eProDas_Error_ClockTooLow`. Further, when the extended divider is operational the minimal main divider is imposed by the value `MinimalMainDividerWithExt` of the machine readable report (see appendix C for full explanation). Again, the non-zero parameter `Forced` will let you do pretty much everything in the limits of technical feasibility.

**Note 1.** As explained in appendix C the operational extended divider consumes a couple of additional instruction cycles contrary to operation without it. Mathematically, it is equivalent to specify parameter `ExtendedDivider` of 1 (divide by 1) or 0 (switch off), but from the eProDas point of view it is not, since the value of 0 saves additional instruction cycles and speeds up execution of periodic actions for a certain amount of time. In order not to bother users with this subtle difference function `SetPeriodicClockRaw` internally changes the `ExtendedDivider` value of 1 into 0, which is always a correct thing to do. Although the change could be done silently, we feel that you need to be informed about this behaviour. Namely, if you examine clock configuration for this particular case at a later time (section 19.7.2) you may notice the discrepancy between your selection of `ExtendedDivider` and the actual setting, which may be confusing.

**Note 2.** Although function `SetPeriodicClockRaw` is probably the least convenient of all functions for setting periodic clock, it is the only function that gives you the total control over periodic clock value. All other functions for setting the clock try to deduce appropriate values for `MainDivider` and `ExtendedDivider` from a more intuitively specified desired frequency and then they call the function `SetPeriodicClockRaw` behind the scenes to actually configure periodic clock.

**If `SetPeriodicClockRaw` cannot set certain clock frequency no other function can do better.**

### 19.7.2 Learning about periodic clock settings

When discussing the functions `SetPeriodicClockFrequency` and `SetPeriodicClockPeriod`, we warned you that when you try to set periodic clock to an impossible value, these two functions set the clock to something possible in a close proximity to your demand. For that matter it is always wise to inspect the resulting periodic clock with the following function.

#### 19.7.2.1 GetPeriodicClock

Prototype in C/C++

```
int eProDas_GetPeriodicClock(unsigned int DeviceIdx,
  eProDasStruct_PeriodicClock &Data);
```

Prototype in Delphi

```
function eProDas_GetPeriodicClock(DeviceIdx: LongWord;
  var Data: TeProDasStruct_PeriodicClock):integer;
```

Prototype in VisualBasic

```
Function eProDas_GetPeriodicClock (ByVal DeviceIdx As UInteger,
  ByRef Data As TeProDasStruct_PeriodicClock) As Integer
```

With this function you retrieve the configuration of the periodic clock. Further, since frequency or period of sampling rate, etc. is often displayed on screen or reported in other ways by many data acquisition applications, this function also retrieves clock information in a form that is suitable for such tasks. So, you may find this function handy even if you are not curious about clock settings.

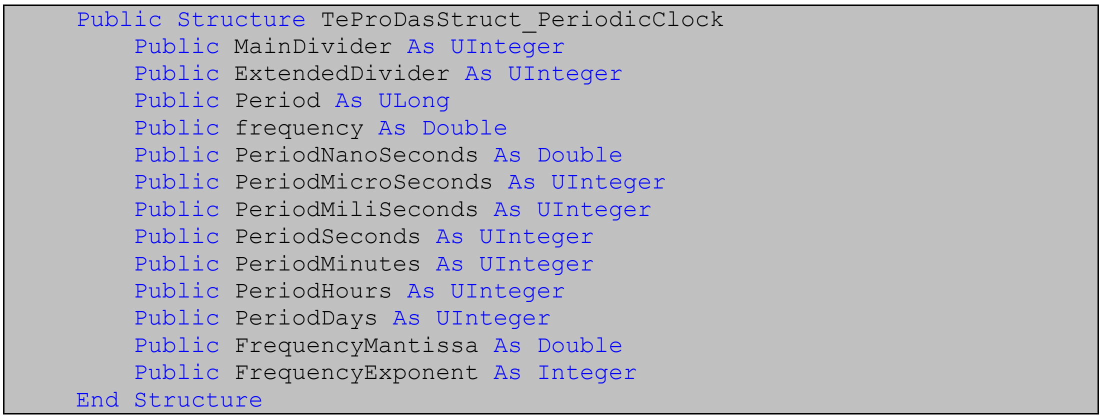
The configuration of periodic clock is retrieved through the parameter `Data`, which is a structure of a type `eProDasStruct_PeriodicClock` that is presented in the following figures.

```
struct eProDasStruct_PeriodicClock {
    unsigned int MainDivider;
    unsigned int ExtendedDividerUsed;
    unsigned int ExtendedDivider;
    __int64 Period;
    double Frequency;
    double PeriodNanoSeconds;
    unsigned int PeriodMicroSeconds;
    unsigned int PeriodMiliSeconds;
    unsigned int PeriodSeconds;
    unsigned int PeriodMinutes;
    unsigned int PeriodHours;
    unsigned int PeriodDays;
    double FrequencyMantissa;
    int FrequencyExponent;
};
```

**Figure 82: The C/C++ definition of the `eProDasStruct_PeriodicClock`.**

```
TeProDasStruct_PeriodicClock = record
    MainDivider: LongWord;
    ExtendedDividerUsed: LongWord;
    ExtendedDivider: LongWord;
    Period: Int64;
    Frequency: Double;
    PeriodNanoSeconds: Double;
    PeriodMicroSeconds: LongWord;
    PeriodMiliSeconds: LongWord;
    PeriodSeconds: LongWord;
    PeriodMinutes: LongWord;
    PeriodHours: LongWord;
    PeriodDays: LongWord;
    FrequencyMantissa: Double;
    FrequencyExponent: Integer;
end;
```

**Figure 83: The Delphi definition of the `eProDasStruct_PeriodicClock`.**



```
    Public Structure TeProDasStruct_PeriodicClock
        Public MainDivider As UInteger
        Public ExtendedDivider As UInteger
        Public Period As ULong
        Public frequency As Double
        Public PeriodNanoSeconds As Double
        Public PeriodMicroSeconds As UInteger
        Public PeriodMiliSeconds As UInteger
        Public PeriodSeconds As UInteger
        Public PeriodMinutes As UInteger
        Public PeriodHours As UInteger
        Public PeriodDays As UInteger
        Public FrequencyMantissa As Double
        Public FrequencyExponent As Integer
    End Structure
```

**Figure 84: The VisualBasic definition of the `eProDasStruct_PeriodicClock`.**

The first five fields of the structure summarize actual settings of periodic clock. Fields `MainDivider` and `ExtendedDivider` hold values of main and extended divider, respectively. When extended divider is operational the field `ExtendedDividerUsed` holds a nonzero value. The length of the resulting period in 12 MHz ticks is reported in `Period` field. Finally, `Frequency` holds the frequency of periodic actions in Hertz.

The remaining fields of the structure summarize the same information in a more suitable way for displaying or reporting. The fields `PeriodDays`, `PeriodHours`, `PeriodMinutes`, `PeriodSeconds`, `PeriodMiliSeconds`, `PeriodMicroSeconds` and `PeriodNanoSeconds` disassemble the period into the self descriptive components. So, it is easy to report on screen something like: the period of clock is 5 hours plus 7 minutes plus 3 seconds… Note that all these fields are of type integer except nanoseconds are reported as a floating point number, which is due to the fact that period of 12 MHz clock does not divide evenly with nanoseconds.

For displaying frequency there exist two fields `FrequencyMantissa` and `FrequencyExponent`. The former always contains number in the range from 0 to 1,000, whereas the latter tells the decimal suffix by which a correct frequency is calculated. For example, when periodic frequency is 25 kHz, `FrequencyMantissa` contains number 25 and `FrequencyExponent` number 3. Similarly, for frequency of 35 nHz, the respective numbers are 35 and -9. This way it is easy to display frequency in a human readable format like 35 nHz or $35 \cdot 10^{-9}$ Hz instead of being forced to output 0.000000035 Hz. See section 25.2 for description of functions that can further assist you with this task.

## 19.8 Using analysis report in your applications

By reading textual report one learns a lot about periodic program. The information may be also used by end users' applications to run correctly or optimally. Of course, applications are not as good at reading descriptive textual outputs as human beings are and for that matter there exist a function `GetParametersOfPeriodicActions` that packs the most useful parameters in a `Data` structure.

### 19.8.1 GetParametersOfPeriodicActions

Prototype in C/C++

```
int eProDas_GetParametersOfPeriodicActions(unsigned int DeviceIdx,
  eProDasStruct_PeriodicParameters &Data);
```

Prototype in Delphi

```
function eProDas_GetParametersOfPeriodicActions(DeviceIdx: LongWord;
  var Data: TeProDasStruct_PeriodicParameters): integer;
```

Prototype in VisualBasic

```
Function eProDas_GetParametersOfPeriodicActions (ByVal DeviceIdx As UInteger,
  ByRef Data As TeProDasStruct_PeriodicParameters) As Integer
```

Upon return the structure `Data` holds the retrieved information. The definition of the structure is as follows.

```
struct eProDasStruct_PeriodicParameters {
  unsigned int DeviceIdx;
  unsigned int ConfigurationStep;

  unsigned int PacketMode;

  unsigned int Warnings;
  unsigned int Errors;
  unsigned int FatalErrors;

  unsigned int OnePeriodSize_IN_Packet;
  unsigned int OnePeriodSize_OUT_Packet;

  unsigned int ActualSize_IN_Packet;
  unsigned int ActualSize_OUT_Packet;

  unsigned int RequestedPeriodsForOne_IN_Packet;
  unsigned int RequestedPeriodsForOne_OUT_Packet;

  unsigned int ActualPeriodsForOne_IN_Packet;
  unsigned int ActualPeriodsForOne_OUT_Packet;

  unsigned int NumberOf_IN_Buffers;
  unsigned int NumberOf_OUT_Buffers;

  unsigned int Duration;

  double MaxFrequency;

  double MaxFrequencyMantissa;
  int MaxFrequencyExponent;
  char MaxFrequencyPrefix;

  unsigned int MinMainDivider;
  unsigned int MinMainDividerWithExt;

  unsigned int DurationWithAD;

  double MaxFrequencyWithAD;

  double MaxFrequencyMantissaWithAD;
  int MaxFrequencyExponentWithAD;
  char MaxFrequencyPrefixWithAD;

  unsigned int MinMainDividerWithAD;
}; //struct eProDasStruct_PeriodicParameters
```

**Figure 85: The C/C++ definition of the `eProDasStruct_PeriodicParameters`.**

```
  TeProDasStruct_PeriodicParameters = record
    DeviceIdx: LongWord;
    ConfigurationStep: LongWord;

    PacketMode: LongWord;

    Warnings: LongWord;
    Errors: LongWord;
    FatalErrors: LongWord;

    OnePeriodSize_IN_Packet: LongWord;
    OnePeriodSize_OUT_Packet: LongWord;

    ActualSize_IN_Packet: LongWord;
    ActualSize_OUT_Packet: LongWord;

    RequestedPeriodsForOne_IN_Packet: LongWord;
    RequestedPeriodsForOne_OUT_Packet: LongWord;

    ActualPeriodsForOne_IN_Packet: LongWord;
    ActualPeriodsForOne_OUT_Packet: LongWord;

    NumberOf_IN_Buffers: LongWord;
    NumberOf_OUT_Buffers: LongWord;

    Duration: LongWord;

    MaxFrequency: Double;
    MaxFrequencyMantissa: Double;
    MaxFrequencyExponent: Integer;
    MaxFrequencyPrefix: Char;

    MinMainDivider: LongWord;
    MinMainDividerWithExt: LongWord;

    DurationWithAD: LongWord;

    MaxFrequencyWithAD: Double;

    MaxFrequencyMantissaWithAD: Double;
    MaxFrequencyExponentWithAD: Integer;
    MaxFrequencyPrefixWithAD: Char;

    MinMainDividerWithAD: LongWord;
end;
```

**Figure 86: The Delphi definition of the `eProDasStruct_PeriodicParameters`.**

The field `DeviceIdx` is there to assist you not to get lost when debugging your applications by forgetting to which device this report belongs to.

The field `ConfigurationStep` reveals at which step of periodic configuration (Figure 55 on page 167, Table 9 on page 168) you are currently located. If the step is not `eProDas_PeriodicStep_Analysis` or greater, then all other fields are meaningless and filled with zeros.

The field `PacketMode` holds the selected mode of USB packets handling according to the Table 10 on page 197.

The fields `Warnings`, `Errors` and `FatalErrors` hold the number of reported warnings, errors of normal severity and fatal errors, respectively.

The field `OnePeriodSize_IN_Packet` specifies how many bytes of data are generated by device during one period of execution. Similarly, `OnePeriodSize_OUT_Packet` reveals the number of host-PC-provided bytes that the device consumes during one period of execution.

```
Public Structure TeProDasStruct_PeriodicParameters
    Public DeviceIdx As UInteger
    Public ConfigurationStep As UInteger

    Public PacketMode As UInteger

    Public Warnings As UInteger
    Public Errors As UInteger
    Public FatalErrors As UInteger

    Public OnePeriodSize_IN_Packet As UInteger
    Public OnePeriodSize_OUT_Packet As UInteger

    Public ActualSize_IN_Packet As UInteger
    Public ActualSize_OUT_Packet As UInteger

    Public RequestedPeriodsForOne_IN_Packet As UInteger
    Public RequestedPeriodsForOne_OUT_Packet As UInteger

    Public ActualPeriodsForOne_IN_Packet As UInteger
    Public ActualPeriodsForOne_OUT_Packet As UInteger

    Public NumberOf_IN_Buffers As UInteger
    Public NumberOf_OUT_Buffers As UInteger

    Public Duration As UInteger

    Public MaxFrequency As Double

    Public MaxFrequencyMantissa As Double
    Public MaxFrequencyExponent As Integer
    Public MaxFrequencyPrefix As Char

    Public MinMainDivider As UInteger
    Public MinMainDividerWithExt As UInteger

    Public DurationWithAD As UInteger

    Public MaxFrequencyWithAD As Double

    Public MaxFrequencyMantissaWithAD As Double
    Public MaxFrequencyExponentWithAD As Integer
    Public MaxFrequencyPrefixWithAD As Char

    Public MinMainDividerWithAD As UInteger
End Structure
```

**Figure 87: The VisualBasic definition of the `eProDasStruct_PeriodicParameters`.**

The fields `ActualSize_IN_Packet` and `ActualSize_OUT_Packet` reveal the sizes of IN and OUT packets, respectively. When certain type of packets is not used, the respective field is set to zero.

The fields `RequestedPeriodsForOne_IN_Packet` and `RequestedPeriodsForOne_OUT_Packet` are copies of your demands regarding the packets' duration (the parameters `IN_Periods` and `OUT_Periodis`, respectively, of the function SetPeriodicMode). Associated are the two fields `ActualPeriodsForOne_IN_Packet` and `ActualPeriodsForOne_OUT_Packet` that hold the duration of each packet's buffer of the appropriate packet type in number of periods of execution. When certain type of packets is not used, the respective field is set to zero.

The fields `NumberOf_IN_Buffers` and `NumberOf_OUT_Buffers` provide physical buffers count for each packet type. When certain type of packets is not used, the respective field is zero; otherwise it is 6, except when the number of IN buffers is reduced to extend the capacity of the constant OUT buffer.

The field `Duration` holds the duration of the whole period of execution (including Entry and Exit routines) in 12 MHz clock periods. To respect the dynamics of AD use the field `DurationWithAD`.

### 19.8.2 Knowing and displaying frequency limit

If your application wants to know the maximal permissible frequency of the periodic program (in Hz) it can learn about it by reading the field `MaxFrequency`. Further, for displaying this figure on the screen, etc. the structure provides the two fields `MaxFrequencyMantissa` and `MaxFrequencyExponent`. There is also a decimal prefix `MaxFrequencyPrefix`, which holds ASCII character that it associated with the value of `MaxFrequencyExponent`.

For example, if `MaxFrequency` holds the value of 12000 (Hz), then `MaxFrequencyMantissa` holds the value of 12 and `MaxFrequencyExponent` holds the value of 3. The `MaxFrequencyPrefix` contains "k" for kilo in this case.

The fields `MaxFrequencyWithAD`, `MaxFrequencyMantissaWithAD`, `MaxFrequencyExponentWithAD` and `MaxFrequencyPrefixWithAD` hold the same information as the respective fields without suffix `WithAD` do, except that this time the dynamics of internal AD converter is fully respected.

The field `MinMainDivider` contains the minimal permissive main divider that assures accurate frequency and jitterless operation; the field `MinMainDividerWithAD` also assures that internal AD conversion is not started prematurely at the beginning of the period.

The field `MinMainDividerWithExt` also respects the peculiarities of the extended divider (appendix C) and assures accurate frequency and jitterless operation when extended divider is operational.

## 19.9 All about data transfer

We are approaching to the main point of this whole chapter: running general-case periodic actions with bidirectional data exchange. So far we have said more or less everything about all preparation steps up to and including the step **Prepare(6)** (Figure 55 on page 167). Now we need to discuss the most important ingredient of the recipe, which is the actual data transfer between your eProDas device and the host PC. This material is essential to successfully *send initial data to device* (the next rectangle after the step **Prepare(6)** in the Figure 55) and to *exchange data with device* during the experiment (the next rectangle after the step **Run(7)** in the Figure 55).

Section 19.4 discusses the important background principles of eProDas's periodic data transfer. Now we are focusing on the functions that your application actually uses to accomplish the task. Let us start with two the most fundamental of them.

### 19.9.1 Sending and receiving periodic packets

Prototypes in C/C++

```
int eProDas_SendPeriodicPacket(unsigned int DeviceIdx, unsigned char *Data);

int eProDas_ReceivePeriodicPacket(unsigned int DeviceIdx, unsigned char *Data);
```

Prototypes in Delphi

```
function eProDas_SendPeriodicPacket(DeviceIdx: LongWord; Data: PByte):integer;

function eProDas_ReceivePeriodicPacket(DeviceIdx: LongWord; Data: PByte):integer;
```

Prototypes in VisualBasic

```
Function eProDas_SendPeriodicPacket (ByVal DeviceIdx As UInteger,
  ByRef Data As Byte) As Integer

Function eProDas_ReceivePeriodicPacket (ByVal DeviceIdx As UInteger,
  ByRef Data As Byte) As Integer
```

Call the function SendPeriodicPacket when you want to send (the next) OUT packet to the device. All you need to specify is the pointer Data to the first byte of the properly formed data buffer according to the principles outlined in the section 19.4 and illustrated in the Figure 69 (page 187). You do not specify the length of the buffer, since the transferred amount always equals the value of the parameter ActualSize_OUT_Packet (section 19.8) and your application will crash if the allocated space of the pointed-to Data buffer is smaller than this size.

Similarly, call the function ReceivePeriodicPacket when you want to receive (the next) IN packet from the device. This time the parameter Data points to the first byte of the buffer to which the data should be written to. The transferred amount always equals the value of the parameter ActualSize_IN_Packet (section 19.8), so you do not have to specify the amount of data to be transferred. Your application will crash if the pointed-to Data buffer is smaller than this size.

The two functions seem rather simple on the outside but internally they are fairly complex. In fact, behind the scenes the whole eProDas stack works in a synergic way to achieve as fast data throughput as feasible. If you understand the workings of these two functions in detail you will find the development of eProDas periodic applications a breeze; however, any misunderstanding will surely bring you into trouble. It is extremely important that you read the following explanations carefully and thoroughly in order not to get mad at eProDas if things do not work as expected.

### 19.9.2 Sending and receiving under the hood

The first important thing to know about sending and receiving is that these two operations work completely asynchronously. When you call one of the above functions your demand is only remembered internally by eProDas device driver and then the function returns to you instantly. The actual data transfer between host PC and eProDas device may happen substantially later when some buffer on the device is ready for the transfer.

In the case of the function SendPeriodicPacket this means that the next transfer happens when some OUT buffer on the device is consumed and declared empty. For the function ReceivePeriodicPacket the next transfer happens when some IN buffer on the device is filled in with generated data and declared full. Note that depending on your periodic program, clock settings and packet delivery configuration, the delay between the demand and the actual completion of the transfer may be as long as several days. This means, for example, that you must not consume the contents of the IN packet immediately after returning from the function ReceivePeriodicPacket. Instead, you must make sure that the operation is indeed completed prior to touching the IN buffer.

The asynchronous nature of the two functions is needed to achieve a decent USB throughput. Namely, the USB bus has fairly high data rates but it is reluctant to send individual packets without noticeable delay between request and the actual transfer. However, when USB bus does decide to finally do some work it is very efficient and it has no problems sending or receiving a large bunch of packets in a short time. For the eProDas application developer this simply means that to achieve high data rates he or she must make sure that there are **always** outstanding OUT and IN demands **waiting in a queue**. The more packets that are waiting to be transferred, the greater **average** transfer rates may be achieved. Therefore, when you call the function to send or receive some packet it returns to you **instantly** (before the actual transfer happens) so that you have a chance to call these functions again for several times to queue several additional packets for a more efficient transmission.

**Note.** Demand for many outstanding packets **does not** mean that you should call these two functions several times to transfer **the same** OUT or IN packet. Prepare more OUT packets or IN buffers in advance and call the two functions each time for **different** i.e. **the next** data packet to the one that you are sending or receiving with the previous call to the very function. You must **never** call these functions more than once for the same packet, if the function **accepted** your request; see below.

### 19.9.3 Status that sending and receiving returns

Especially when doing periodic data transfer it is essential to check the error code that both currently discussed functions return. This time the returned status is not a mere diagnostic tool but a truly needed status without which **you cannot** implement your periodic application.

Let us start with the ordinary errors that are of a diagnostic nature. If you demand any data transfer before successfully calling the function `PreparePeriodicActions` both functions return with the error `eProDas_Error_InvalidPeriodicSequence`.

Similarly, if you demand transfer into a non-existing direction the error `eProDas_Error_InvalidMode` results. This happens for example, if your periodic program only generates IN packets and it does not consume any OUT ones (like when selecting periodic mode `PeriodicMode_InAtEnd`; section 19.6), but you nonetheless try to send some data to the device with the function `SendPeriodicPacket`.

**Now about a bit more important situation.** Whenever eProDas encounters that something is wrong with any delivered periodic USB packet, it remembers this fact internally. From that moment on, it stops all further periodic data transfers and consequently the two transferring functions always return with `eProDas_Error_DataCorruptionError`. It is important to know, that this error is not relevant to the packet that you are currently trying to send or receive. Instead it merely signals that there was(were) error(s) **somewhere in the past** and that everything that you can do about it is to stop (abort) periodic actions and try again. You cannot know when the error occurred so generally you should not trust the IN data that was delivered to you in the past unless you are sure that some delivery has already completed before the error occurred.

The erroneous USB packets are due to errors on USB bus and are therefore not your responsibility but a mere coincidence that happens from while to while due to noise and interferences. If eProDas was an ordinary piece of hardware like hard disk it would repeat the transfer in case of errors. However, doing so would require noticeable performance decrease of periodic actions due to more CPU intensive handling of USB transactions. In likely cases there should be no data corruption errors, so eProDas decides to squeeze as much performance out of PIC chip as possible in the general errorless case but it can merely abort the thing altogether in those misfortunate but hopefully rare periodic cases where errors do happen.

**Now about the most important return code.** eProDas device driver can do internal bookkeeping of one uncompleted transfer for each OUT as well as for each IN buffer on the device (PIC). Whenever you initiate the transfer by calling one of the two functions, the device driver establishes to which physical buffer the transfer is targeted. Then it checks whether there already exists previous uncompleted transfer demand for this buffer (not only for this direction, but especially for this buffer). If this is not the case the driver accepts your requests and returns with a success. This means that the packet will be delivered when the device is ready for it.

However, if the demand is already scheduled for the very same physical buffer, device driver rejects your demand with the error `eProDas_Error_NotEnoughResources`. It is important to know, that this is **not** an error but merely an indicator about too many concurrent transfers in progress. You should repeat the transfer demand for **the same** packet again at a later time, when the device may be ready.

#### 19.9.3.1 An example

Suppose that we want to run some periodic program that produces two bytes of IN data during each period of execution (say, one AD result). We intend to run this program at a frequency of 1 kHz and we utilize full sized USB packets of 64 bytes: eProDas device delivers one packet per 32 executions of our program, which happens once per 32 ms at 1 kHz clock.

Let us allocate a buffer for incoming IN packets. We also need to provide a pointer to be able to walk along this buffer when demanding transfer requests. The following code achieves both of these tasks.

```
1.  const unsigned int NoPackets = 16;
2.  unsigned char IN_Pipe[ActualSize_IN_Packet * NoPackets];
3.  unsigned char *PtrIN = IN_Pipe;
4.  int Status;
```

**Figure 88: Allocation of IN buffer and initialization of a pointer for walking along the buffer.**

The first line symbolically shows that we want to receive 16 packets in total during the whole experiment.

The second line allocates IN buffer that is going to receive the data. The size of this buffer simply equals the product of IN packet size times the number of packets that we want to receive. Note the name of the buffer `IN_Pipe`, which stresses that this buffer should not be treated as an isolated set of packets but it is semantically more appropriate to look at as a contiguous IN pipe that just delivers the data in packets for technical reasons.

The third line declares a pointer `PtrIN` and initializes it to the starting address of the buffer.

The fourth line declares the well known `Status` variable, which is truly obligatory in this example.

Now, we start periodic actions. The first results will be ready only 32 ms later, but we should demand their transfer in advance to fully exploit the asynchronous nature of eProDas's periodic transfer subsystem. Since we have nothing else to do, we call the function `ReceivePeriodicPacket` immediately after the start of periodic actions. The call should look something like the following line of code.

```
5.  Status = ReceivePeriodicPacket(DeviceIdx, PtrIN);
```

**Figure 89: Demanding of IN data transfer.**

All we have to do is to specify the address of the buffer, where the data should be delivered. We merely provide the value of the pointer `PtrIN`, which was previously initialized to the beginning of the whole buffer.

eProDas checks to which physical buffer this transfer is associated with. We call the function `ReceivePeriodicPacket` for the first time, so it naturally follows that the transfer targets the first IN buffer on the device. This is the first transfer demand for this particular buffer so no previous data transfer request in progress is associated with it. eProDas accepts our request and returns with success.

Now it is our duty to check the outcome of the request. The code could but it does not have to look something like the following code fragment.

```
 6. if(Status == eProDas_Error_NotEnoughResources) {
 7.   //request rejected, we should repeat it later. This is NOT an error
    }
 8. else if(Status != eProDas_Error_SUCCESS) {
 9.   //something is seriously wrong. Inspect error code in detail to get the cause
10.   Abort_Or_Recover_Somehow();
    }
    else {
11.   //Ok, the request was accepted, advance the pointer
12.   PtrIN += ActualSize_IN_Packet;
    }
```

**Figure 90: Checking the status of the transfer demand.**

Line 6 checks whether the returned status is `eProDas_Error_NotEnoughResources`. This return code does not signal error but a temporal congestion so it should be treated separately. Related line 7 symbolically shows that here is the place to do tasks, which we should do when the congestion arises. Perhaps, we could calculate how long we can fall asleep and then we would do the sleeping before trying again with our request.

Line 8 checks whether some true error like data corruption happened. There is a little chance to do anything else but to abort the whole actions altogether. Anyway, lines 9 and 10 are there as a placeholder for a rescue team.

If the function returns with a success, line 12 must take care to advance the pointer, so that the next request will specify different target address. As you can see, we merely add the IN packet length to the current address that the pointer points to.

The important point here is that in the case of a success the first 64 bytes of our buffer are under the control of the eProDas periodic transfer subsystem and we must not touch this area until the data magically appears there, when eProDas device manages to fill in the first buffer. The data will be written to our buffer behind the scenes and without any further intervention from our side.

The execution of the lines from 5 to 12 happens almost instantly (or better, on the order of microseconds) so there are still slightly less than 32 ms until the first chunk of data is ready. Despite the fact that the transfer cannot be done just yet our application can continue doing other tasks.

Well, reception of these `NoPackets` data packets is all that we need to do, so we call the function `ReceivePeriodicPacket` one more time; the call could be done in a program loop, so the same line 4 (Figure 89) could be used for the task. Since the pointer has been previously advanced (line 12) the transfer demand properly targets the next 64 bytes of our buffer.

eProDas driver establishes that this time the demand is for the second IN buffer on the device, which has no already scheduled requests yet, so the demand is accepted and the function returns immediately. Since the device has six IN buffers (it could also be 4 or 2; section 19.6.4) we can repeat the request for four additional times and all requests will be accepted immediately. Each call should specify different `Data` pointer to deliver each packet to its own place. Six IN transfers are now demanded which means that the eProDas device driver has one transfer request for each physical IN buffer on the device. We consume only a couple of microseconds to schedule all six transfers, which means that there are still slightly less than 32 ms before the first results are ready for us.

Now, let us try to call the function `ReceivePeriodicPacket` one more time. The next physical IN buffer is now again the first one (rollover from the sixth one which was the target of the last successful request). The driver has already an accepted transfer demand for this buffer so it refuses this very last request. The function returns immediately with the error `NotEnoughResources`.

This error indicates that internally nothing happened. The next to be issued request will still be targeting the first IN buffer; if you repeat the function call immediately, the driver will again establish that the associated physical buffer is the first one, which is busy, so the demand would be rejected again.

The important point here is that your application must notice the rejection of the request so that the next time it demands delivery of **the same** packet, i.e. it **does not** advance the pointer until the request succeeds, as the code in the Figure 90 does in a proper way.

Okay, repeating the failed request over and over again is a boring thing to spend our time on, so the application should now really do something else (like displaying previously delivered results, if there were any). If nothing better comes along, we should take a sleep until the first packet is delivered to us. As we previously calculated, one packet is delivered in 32 ms, so the call `eProDas_Sleep(32)` is a good choice (section 18.2.3) as a replacement of line 7 in the Figure 90.

After 32 ms passes (actually it is slightly more; you should always take USB delay into account too), **the first** IN packet from **the first** physical IN buffer should be delivered to us; the data magically appear in the first 64 bytes of the buffer `IN_Pipe`. Accordingly, the first physical IN buffer does not have any associated transfer requests, so when we call the function `ReceivePeriodicPacket` again, our demand for delivery of **the seventh** packet from **the first** IN buffer would be accepted.

After another 32 ms or so pass by, **the second** IN packet would be delivered from **the second** physical IN buffer and the function `ReceivePeriodicPacket` would be willing to accept a request for **the eight** packet from **the second** buffer. Now the story goes on and on until we decide to stop the thing altogether.

#### 19.9.3.2 Conclusions

The above discussion and example would be very similar if we were talking about transfer of OUT packets. The mechanism and semantics are the same for both directions, only the source of and destination for data are swapped. OUT and IN directions are independent of each other, so if there is no room to accept the new IN request there may be enough resources to accept the new OUT request and vice versa.

Note however, that sometimes you need to demand requests for one direction faster than for the other one. Suppose that the just discussed example program also consumed OUT in addition to the IN packets. If the program reads 4 bytes of OUT packet during each execution, it would need a new OUT packet after each 16 ms but it would still deliver one IN packet in 32 ms.

The task of your application during periodic action is that neither OUT nor IN pipe have too little requests in queue. **Do not be afraid** of receiving the error `eProDas_Error_NotEnoughResources`, since this **is not** an error indicator but it is semantically rather a synchronization mechanism. Your application should make sure that it schedules packet requests more frequently than the device can consume them so that asynchronous nature of eProDas transfer subsystem can be exploited efficiently. The error `eProDas_Error_NotEnoughResources` is your friend which **helps you** not to hog the resources by signalling to you “okay, you have done enough requests for a while, take some rest”.

### 19.9.4 Other functions for sending and receiving

Technically, the just described functions SendPeriodicPacket and ReceivePeriodicPacket are all that you need regarding data transfer to implement any eProDas periodic application. Still, there are some typical situations of usage of these functions that we find ourselves coding over and over again. To spare your effort, eProDas possesses additional transfer functions that you may use instead of or in addition to the bare send and receive. However, keep in mind that all the soon to be described extensions rely internally on either SendPeriodicPacket or ReceivePeriodicPacket, so they behave accordingly and the principles of operation as well as returned error codes are intimately connected with the previously described semantics.

#### 19.9.4.1 Sending and receiving with automatic pointer advancing

Prototypes in C/C++

```
int eProDas_SendPeriodicPacket_Advance(unsigned int DeviceIdx,
  unsigned char *&Data);

int eProDas_ReceivePeriodicPacket_Advance(unsigned int DeviceIdx,
  unsigned char *&Data);
```

Prototypes in Delphi

```
function eProDas_SendPeriodicPacket_Advance(DeviceIdx: LongWord;
  var Data: PByte):integer;

function eProDas_ReceivePeriodicPacket_Advance(DeviceIdx: LongWord;
  var Data: PByte):integer;
```

Prototypes in VisualBasic

```
Function eProDas_SendPeriodicPacket_Advance (ByVal DeviceIdx As UInteger,
  ByRef Data As Byte) As Integer

Function eProDas_ReceivePeriodicPacket_Advance (ByVal DeviceIdx As UInteger,
  ByRef Data As Byte) As Integer
```

Function SendPeriodicPacket_Advance calls the function SendPeriodicPacket internally, as you would do it by yourself. Then it checks the status that the latter returns and if the status is eProDas_Error_SUCCESS, the pointer Data is advanced for the ActualSize_OUT_Packet number of bytes. Finally, the function returns the same error code as the function eProDas_SendPeriodicPacket internally produced.

A completely analogous discussion holds for the function ReceivePeriodicPacket_Advance.

Whenever you work with data pipes, where some pointer needs to be advanced in the case of an accepted transfer demand (as the Figure 90 demonstrates), you may find these two functions handy. Also, some development environments do no possess such rich and orthogonal pointer arithmetic as the C/C++ programming language does, so in such cases, these two functions are your only choice.

### 19.9.4.2 Sending and receiving with automatic pointer advancing and stopping at the end

Prototypes in C/C++

```
int eProDas_SendPeriodicPacket_AdvanceWithStop(unsigned int DeviceIdx,
  unsigned char *&Data, unsigned char *StopPtr, unsigned char *PostStopData);

int eProDas_ReceivePeriodicPacket_AdvanceWithStop(unsigned int DeviceIdx,
  unsigned char *&Data, unsigned char *StopPtr);
```

Prototypes in Delphi

```
function eProDas_SendPeriodicPacket_AdvanceWithStop(DeviceIdx: LongWord;
  var Data: PByte; StopPtr: PByte; PostStopData: PByte):integer;

function eProDas_ReceivePeriodicPacket_AdvanceWithStop(DeviceIdx: LongWord;
  var Data: PByte; StopPtr: PByte):integer;
```

Prototypes in VisualBasic

```
eProDas_SendPeriodicPacket_AdvanceWithStop


eProDas_ReceivePeriodicPacket_AdvanceWithStop
```

Every our data buffer has its end and your buffers are probably not much different in this respect. Often you implement excitation by providing a limited number of excitation data. After these are consumed you continue excitation with some constant values that make sense in your particular case. Such situations frequently arise especially when you observe some transient response of a system: you intensively excite the system in the first phase of the experiment and then you stop the excitation (like outputting voltage of 0 V) but you continue to measure the observed response (variable).

The function `SendPeriodicPacket_AdvanceWithStop` mimics this behaviour. As a first step the function checks whether the pointer `Data` is less than the pointer `StopPtr`. If so, it calls the function `SendPeriodicPacket_Advance` with pointer `Data` as the argument; the pointer `Data` is advanced in the case of an accepted request. However, if the pointer `Data` is greater than or equal to `StopPtr`, the bare function `SendPeriodicPacket` is called with the parameter `PostStopData` as the argument.

The bottom line: eProDas will use the “main” buffer pointed to by the pointer `Data`, until all its content is consumed. After that it will always send the contents of the buffer `PostStopData` to the device no matter how many times you call the function `SendPeriodicPacket_AdvanceWithStop`.

**Note 1.** If the packet `PostStopData` contains all zeros, you can specify null pointer for this parameter and eProDas will provide its own zero-filled buffer for the task.

**Note 2.** The function `SendPeriodicPacket_AdvanceWithStop` is completely stateless. The decision about which buffer to send to the device is resolved during each function call solely by comparing pointers `Data` and `StopPtr`. If you reach the end of the buffer and then reinitialize pointer `Data` to some value that is less than `StopPtr`, the function will again use the values from the main buffer, since it does not remember internally that the end of the buffer condition has already been reached.

**Note 3.** The pointer `StopPtr` must not point to the last byte in the buffer but to the one past the last. This is the usual arrangement of pointers and it should be familiar to you if you have any experiences with data buffers. The next code fragment shows the usual initialization of buffer’s pointers.

```
1.  const unsigned int Size = 256;
2.  unsigned char Buffer[Size];
3.  unsigned char *Data = Buffer;
4.  unsigned char *StopPtr = Buffer+Size;
```

**Figure 91: Start and stop pointers of some buffer.**

Line 1 symbolically hints the capacity of the buffer. The (static in this case) allocation happens in line 2. Line 3 initializes pointer `Data` to the beginning of the buffer. Similarly, line 4 initializes pointer `StopPtr` to the one past the end of the buffer, which is due to the fact that `Buffer`+0 points to the first element, `Buffer`+1 to the second and `Buffer`+(`Size`-1) to the last one. That is the proper way to get prepared for the usage of the function `SendPeriodicPacket_AdvanceWithStop`.

As you can see from the definition, the function `ReceivePeriodicPacket_AdvanceWithStop` is not completely analogous to the function `SendPeriodicPacket_AdvanceWithStop`, since there is no parameter `PostStopData`. The working is as follows. As long as the parameter `Data` is less than the `StopPtr`, the function `ReceivePeriodicPacket_Advance` is called internally with pointer `Data` as the argument; upon accepted request the pointer `Data` is advanced. When `Data` becomes equal to or greater than `StopPtr` the function simply returns with an error `eProDas_Error_EndOfBufferReached`.

If the only purpose of your periodic program is to collect enough data to fill in the buffer `Data`, then the error `eProDas_Error_EndOfBufferReached` informs you that there are enough requests in progress to accomplish the task so you can start thinking about stopping the periodic execution. Again, **do not be afraid** of this error since it is more an information about successful submission of all intended transfer requests than it is an indicator of some error condition.

### 19.9.4.3 Sending and receiving with buffer circulation

Prototypes in C/C++

```
int eProDas_SendPeriodicPacket_Circulate(unsigned int DeviceIdx,
  unsigned char *&Data, unsigned char *StartPtr, unsigned char *StopPtr);

int eProDas_ReceivePeriodicPacket_Circulate(unsigned int DeviceIdx,
  unsigned char *&Data, unsigned char *StartPtr, unsigned char *StopPtr);
```

Prototypes in Delphi

```
function eProDas_SendPeriodicPacket_Circulate(DeviceIdx: LongWord; var Data: PByte;
  StartPtr: PByte; StopPtr: PByte):integer;

function eProDas_ReceivePeriodicPacket_Circulate(DeviceIdx: LongWord;
  var Data: PByte; StartPtr: PByte; StopPtr: PByte):integer;
```

Prototypes in VisualBasic

```
eProDas_SendPeriodicPacket_Circulate


eProDas_ReceivePeriodicPacket_Circulate
```

Instead of running through your buffer once, you may want to repeat it indefinitely. Such situation arises when you are implementing some sort of signal generator, where you synthesize the same pattern over and over again. Function `SendPeriodicPacket_Circulate` is there for the task and it works as follows.

First, the function `SendPeriodicPacket_Advance` is called internally with parameter `Data` as an argument; `Data` is being advanced if the request is accepted. Next, if `Data` becomes greater than or equal to the `StopPtr`, it is being reinitialized to `StartPtr`, which should point to the start of the buffer.

**Note.** The pointer `StopPtr` must not point to the last byte in the buffer but to the one past the last. Please, see the discussion in the previous section.

As you can see there also exists a function `ReceivePeriodicPacket_Circulate` for circulated reception of packets, which may seem a bit odd, since we usually do not want to overwrite the received data with the now one over and over again. Nonetheless, there are plenty of opportunities to utilize this function in a good way.

Your host PC can chew data much faster than eProDas can generate. So the former can analyze the contents of some buffer in a fraction of a time that is needed for obtaining the very data. If your intention is not to save the original data but to e.g. extract some statistics out of, say, gigabytes worth of measurements, you do not have to pre-allocate gigabyte-large buffer. Instead, pre-allocate a relatively small buffer like for 18 IN packets if your device uses 6 IN buffers. Then, as new data arrives analyze it immediately and extract the needed statistics, after which the buffer can be safely overwritten with the new data. The experiment may repeat for decades if needed.

The second likely scenario of usage is connected to the utilization of the previously discussed function `ReceivePeriodicPacket_AdvanceWithStop`. Suppose that you want to collect some data that you do not wish to overwrite but after doing it so, you do not want to stop the execution of periodic program. In order for eProDas not to stop with buffer overrun and wait for more receive requests, you must continue supplying demands for reception of IN packets, which you do not even intend to look at.

For that matter you can use the function `ReceivePeriodicPacket_Circulate` in the following way. Allocate two separate buffers, one for reception of true data and the other one for temporal storage of unneeded packets. The latter should be just large enough to assure that no two receive requests in progress could target the same buffer area; as a rule of thumb, this buffer should accept three times more packets than there is active physical IN buffers on the device (section 19.6.4).

At the beginning of the experiment use the function `ReceivePeriodicPacket_AdvanceWithStop` to receive true data to the data buffer. As soon as this function signals that you have provided enough requests by returning with the error code `eProDas_Error_EndOfBufferReached`, use the function `ReceivePeriodicPacket_Circulate` to continue the eternal IN packets flow into the temporal buffer.

#### 19.9.4.4 Sending and receiving with buffer circulation and rollover count

Prototypes in C/C++

```
int eProDas_SendPeriodicPacket_CirculateCount(unsigned int DeviceIdx,
  unsigned char *&Data, unsigned char *StartPtr, unsigned char *StopPtr,
  unsigned int &Counter);
```

```
int eProDas_ReceivePeriodicPacket_CirculateCount(unsigned int DeviceIdx,
  unsigned char *&Data, unsigned char *StartPtr, unsigned char *StopPtr,
  unsigned int &Counter);
```

Prototypes in Delphi

```
function eProDas_SendPeriodicPacket_CirculateCount(DeviceIdx: LongWord;
  var Data: PByte; StartPtr: PByte; StopPtr: PByte; var Counter: LongWord):integer;
```

```
function eProDas_SendPeriodicPacket_CirculateManyCount(DeviceIdx: LongWord;
  var Data: PByte; StartPtr: PByte; StopPtr: PByte; var Counter: LongWord):integer;
```

Prototypes in VisualBasic

```
eProDas_SendPeriodicPacket_Circulate
```

```
eProDas_ReceivePeriodicPacket_Circulate
```

This is a slight variation of the previous two functions, where the value of the parameter `Counter` is incremented each time that a buffer rollover occurs.

#### 19.9.4.5 Submitting several send requests at a time

Prototypes in C/C++

```
int eProDas_SendPeriodicPacket_AdvanceMany(unsigned int DeviceIdx,
  unsigned char *&Data);

int eProDas_SendPeriodicPacket_AdvanceManyWithStop(unsigned int DeviceIdx,
  unsigned char *&Data, unsigned char *StopPtr, unsigned char *PostStopData);

int eProDas_SendPeriodicPacket_CirculateMany(unsigned int DeviceIdx,
  unsigned char *&Data, unsigned char *StartPtr, unsigned char *StopPtr);

int eProDas_SendPeriodicPacket_CirculateManyCount(unsigned int DeviceIdx,
  unsigned char *&Data, unsigned char *StartPtr, unsigned char *StopPtr,
  unsigned int &Counter);
```

Prototypes in Delphi

```
function eProDas_SendPeriodicPacket_AdvanceMany(DeviceIdx: LongWord;
  var Data: PByte):integer;

function eProDas_SendPeriodicPacket_AdvanceManyWithStop(DeviceIdx: LongWord;
  var Data: PByte; StopPtr: PByte; PostStopData: PByte):integer;

function eProDas_SendPeriodicPacket_CirculateMany(DeviceIdx: LongWord;
  var Data: PByte; StartPtr: PByte; StopPtr: PByte):integer;

function eProDas_SendPeriodicPacket_CirculateManyCount(DeviceIdx: LongWord;
  var Data: PByte; StartPtr: PByte; StopPtr: PByte; var Counter: LongWord):integer;
```

Prototypes in VisualBasic

```
eProDas_SendPeriodicPacket_AdvanceMany


eProDas_SendPeriodicPacket_AdvanceManyWithStop


eProDas_SendPeriodicPacket_CirculateMany


eProDas_SendPeriodicPacket_CirculateManyCount
```

Functions with literal `Many` in their names operate in a precisely the same way as their respective `Many` -less counterparts do, except that they repeat certain request as long as they are not rejected. For example, if eProDas periodic transfer engine can accept three additional send requests in this moment, then any of these functions would have the same result as explicitly using their respective counterparts for three times.

Before calling any of these functions make sure that you have enough OUT packets prepared in advance; *enough* means something like two times the number of physical OUT buffers. Generally, one additional packet per physical OUT buffer should be enough, but imagine that a burst USB delivery just happened during some submission so before all submissions are over there appears new room for additional submissions. These functions simply **do not stop** until they face the well known error `eProDas_Error_NotEnoughResources` (which is not an error, as we described it previously).

#### 19.9.4.6 Submitting several receive requests at a time

Prototypes in C/C++

```
int eProDas_ReceivePeriodicPacket_AdvanceMany(unsigned int DeviceIdx,
  unsigned char *&Data);

int eProDas_ReceivePeriodicPacket_AdvanceManyWithStop(unsigned int DeviceIdx,
  unsigned char *&Data, unsigned char *StopPtr);

int eProDas_ReceivePeriodicPacket_CirculateMany(unsigned int DeviceIdx,
  unsigned char *&Data, unsigned char *StartPtr, unsigned char *StopPtr);

int eProDas_ReceivePeriodicPacket_CirculateManyCount(unsigned int DeviceIdx,
  unsigned char *&Data, unsigned char *StartPtr, unsigned char *StopPtr,
  unsigned int &Counter);
```

Prototypes in Delphi

```
function eProDas_ReceivePeriodicPacket_AdvanceMany(DeviceIdx: LongWord;
  var Data: PByte):integer;

function eProDas_ReceivePeriodicPacket_AdvanceManyWithStop(DeviceIdx: LongWord;
  var Data: PByte; StopPtr: PByte):integer;

function eProDas_ReceivePeriodicPacket_CirculateMany(DeviceIdx: LongWord;
  var Data: PByte; StartPtr: PByte; StopPtr: PByte):integer;

function eProDas_ReceivePeriodicPacket_CirculateManyCount(DeviceIdx: LongWord;
  var Data: PByte; StartPtr: PByte; StopPtr: PByte; var Counter: LongWord):integer;
```

Prototypes in VisualBasic

```
eProDas_ReceivePeriodicPacket_AdvanceMany


eProDas_ReceivePeriodicPacket_AdvanceManyWithStop


eProDas_ReceivePeriodicPacket_CirculateMany


eProDas_ReceivePeriodicPacket_CirculateManyCount
```

As we can see, there also exists a whole bunch of `Many`-full receive functions. They work analogously to the functions in the previous section, so there is not much to be said about them.

However, note that the function `ReceivePeriodicPacket_AdvanceManyWithStop` may not be able to hog the receive queue if the end of the buffer is reached before enough requests are submitted.

### 19.9.5 Other functions related to data transfer

Submission of transfer requests may seem everything that you will ever need when programming eProDas periodic applications, but this is far away from the truth. Now it is the time to introduce some useful functions that are related to the so far discussed data transfer.

#### 19.9.5.1 Uncompleted periodic transactions

Prototype in C/C++

```
int eProDas_UncompletedPeriodicSends(unsigned int DeviceIdx,
  unsigned int &UncompletedPeriodicSends);

int eProDas_UncompletedPeriodicReceives(unsigned int DeviceIdx,
  unsigned int &UncompletedPeriodicReceives);
```

Prototype in Delphi

```
function eProDas_UncompletedPeriodicSends(DeviceIdx: LongWord;
  var UncompletedPeriodicSends: LongWord):integer;

function eProDas_UncompletedPeriodicReceives(DeviceIdx: LongWord;
  var UncompletedPeriodicReceives: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_UncompletedPeriodicSends (ByVal DeviceIdx As UInteger,
  ByRef UncompletedPeriodicSends As UInteger) As Integer

Function eProDas_UncompletedPeriodicReceives (ByVal DeviceIdx As UInteger,
  ByRef UncompletedPeriodicReceives As UInteger) As Integer
```

We have seen that periodic transfer is asynchronous in nature, which means that the functions for submitting transfer requests return control to our application before the transfer is done. In many cases we would like to process the received data during the experiment, like when displaying the results on screen in real-time. To do that we must know which transfer requests has already completed and which are still in progress.

Upon successful return function `UncompletedPeriodicSends` sets it's the last parameter to the number of not yet completed send requests, i.e. the submitted OUT packets that are not delivered to the device yet.

Similarly, function `UncompletedPeriodicReceives` reports about not yet completed receive requests, i.e. the submitted IN packet requests that are not delivered to your application yet.

Thus, if upon return the parameter `UncompletedPeriodicReceives` holds the number of 2, this means that two the last IN packets that were demanded to be received from the device are not in place yet. Everything else is completed and you can analyze it immediately.

### 19.9.5.2 Blocking application until something completes

Prototypes in C/C++

```
int eProDas_WaitAnySendPeriodicTransaction(unsigned int DeviceIdx);

int eProDas_WaitAnyReceivePeriodicTransaction(unsigned int DeviceIdx);

int eProDas_WaitAnyPeriodicTransaction(unsigned int DeviceIdx);
```

Prototypes in Delphi

```
function eProDas_WaitAnySendPeriodicTransaction(DeviceIdx: LongWord):integer;

function eProDas_WaitAnyReceivePeriodicTransaction(DeviceIdx: LongWord):integer;

function eProDas_WaitAnyPeriodicTransaction(DeviceIdx: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_WaitAnySendPeriodicTransaction (ByVal DeviceIdx As UInteger)
  As Integer

Function eProDas_WaitAnyReceivePeriodicTransaction (ByVal DeviceIdx As UInteger)
  As Integer

Function eProDas_WaitAnyPeriodicTransaction (ByVal DeviceIdx As UInteger)
  As Integer
```

Imagine the situation, where our application establishes that all buffers are full and that it already displayed all the results (and finished with all other things that it had to do) so now it can merely wait for the eProDas system to become ready to accept the next send or receive request.

These functions with self descriptive names block application execution until the appropriate event happens: at least one OUT transaction completes, at least one IN transaction completes or at least one of the OUT or IN transactions completes, respectively.

**Note 1.** These functions implement the so called passive waiting where your application does not consume CPU time (by checking transactions in a program loop), but your program (thread) relinquishes its CPU interval to other potential running applications. This is a fair thing to do when you have nothing else to do. And it is the only way to stop Windows's task manager from displaying 100 % CPU utilization at all times although you only process one packet per minute.

**Note 2.** These functions are not suitable to use when data rates are extremely high, or better, when you need to deliver one or more packets on a millisecond basis or faster. Windows scheduler is simply too coarse for such tasks. In such cases you need to resort to the active waiting and hog your CPU 100 %.

### 19.9.5.3 Blocking with timeout

Prototypes in C/C++

```
int eProDas_WaitAnySendPeriodicTransactionWithTimeOut(unsigned int DeviceIdx,
  unsigned int MiliSeconds, unsigned int &TimeOut);

int eProDas_WaitAnyReceivePeriodicTransactionWithTimeOut(unsigned int DeviceIdx,
  unsigned int MiliSeconds, unsigned int &TimeOut);

int eProDas_WaitAnyPeriodicTransactionWithTimeOut(unsigned int DeviceIdx,
  unsigned int MiliSeconds, unsigned int &TimeOut);
```

Prototypes in Delphi

```
function eProDas_WaitAnySendPeriodicTransactionWithTimeOut(DeviceIdx: LongWord;
  MiliSeconds: LongWord; var TimeOut: LongWord):integer;

function eProDas_WaitAnyReceivePeriodicTransactionWithTimeOut(DeviceIdx: LongWord;
  MiliSeconds: LongWord; var TimeOut: LongWord):integer;

function eProDas_WaitAnyPeriodicTransactionWithTimeOut(DeviceIdx: LongWord;
  MiliSeconds: LongWord; var TimeOut: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_WaitAnySendPeriodicTransactionWithTimeOut
  (ByVal DeviceIdx As UInteger, ByVal MiliSeconds As UInteger,
  ByRef TimeOut As UInteger) As Integer

Function eProDas_WaitAnyReceivePeriodicTransactionWithTimeOut
  (ByVal DeviceIdx As UInteger, ByVal MiliSeconds As UInteger,
  ByRef TimeOut As UInteger) As Integer

Function eProDas_WaitAnyPeriodicTransactionWithTimeOut
  (ByVal DeviceIdx As UInteger, ByVal MiliSeconds As UInteger,
  ByRef TimeOut As UInteger) As Integer
```

These functions extend the previously described application blocking with the possibility of operation timeout. Namely, when you block your application you essentially lose the control of execution for as long as something finally completes. Occasionally you would want to reacquire the control even if something does not complete for a long time. For example, you may want to respond to an event, like pressed key on a keyboard. This is what these extended functions are about.

With the parameter `MiliSeconds` you specify at most how many milliseconds you are willing to be blocked. If anything appropriate completes anywhere in-between you application regains the control immediately, i.e. before the timeout period expires. Otherwise you are back on track after the specified amount of time passes by.

By checking the parameter `TimeOut` you can learn about the outcome; the non-zero value of it indicates that nothing appropriate completes and that we have a timeout condition, otherwise at least one of the transactions that you were waiting for have completed and there is now some work to do.

## 19.10 Sending initial data to the device

As soon as periodic actions start eProDas device begins to excite outputs (unless your periodic program does not do any form of excitation, of course). In the cases where you use OUT packets, you must send at least one OUT packet to the device **before** periodic actions are started or else buffer underrun would happen immediately upon the beginning of periodic activities. In order not to be tempted to skip this step, the function `StartPeriodicActions` refuses to do the job if not at least one OUT packet has been sent to the device before the function call.

Unless you have a solid reason not to do it (like implementation of a control system instead of an ordinary signal generator), we recommend you to fill in all `NumberOf_OUT_Buffers` (section 19.8.1) OUT buffers that are available, at least in the cases where periodic actions are run at high frequencies, to minimize the risk of buffer underruns. High frequencies in this case means that on average one OUT packet must be delivered to the device in a couple of milliseconds or faster.

To send one or more OUT packets to the device, you can use the most suitable function(s) from the already described `SendPeriodicPacket` family of functions. Let us take a look at the following code fragment.

```
1.   const unsigned int DeviceIdx=0;
2.   int Status;

3.   eProDasStruct_PeriodicParameters PeriodicParams;
4.   eProDas_GetParametersOfPeriodicActions(DeviceIdx, PeriodicParams);

5.   for(int i=0; i < PeriodicParams.NumberOf_OUT_Buffers; i++) {
6.     Status = eProDas_SendPeriodicPacket(DeviceIdx, AddressOf_OUT_Packet(i));
     }
```

**Figure 92: Boilerplate code for sending initial OUT packets.**

Lines 3 and 4 acquire parameters of periodic actions (section 19.8.1) and save them into the variable `PeriodicParams`. Lines from 5 to 6 constitute a program loop that repeats as many times as there are physical OUT buffers on the device. The actual demand for the transfer is symbolically shown in the line 6.

Although this seems all that we can do at this stage, we may in fact do slightly more. In addition to filling in all OUT buffers, we could also send an additional request for each OUT buffer since eProDas driver now has nothing to do, but it could instead hold internally one request per buffer. So, in the most critical cases you can pre-demand twice as much OUT transfers as there are physical OUT buffers on the device. Half of these requests will be delivered to the device before periodic actions are started and the other ones will find its way to the device, when the first set of OUT buffers is slowly consumed during the progress.

Note that until the first half of OUT packets is actually delivered, you cannot successfully submit request for the second half of them. So, you have to go through the tedious task of checking the error code or uncompleted transactions and/or blocking of the application during the process. Fortunately, there exists a special function that goes through these troubles for you.

### 19.10.1 SendPeriodicPacket_AdvanceMany_FillBuffersBeforeStart

Prototypes in C/C++

```
int eProDas_SendPeriodicPacket_AdvanceMany_FillBuffersBeforeStart
   (unsigned int DeviceIdx, unsigned char *&Data);
```

Prototypes in Delphi

```
function eProDas_SendPeriodicPacket_AdvanceMany_FillBuffersBeforeStart
   (DeviceIdx: LongWord; var Data: PByte):integer;
```

Prototypes in VisualBasic

```
Function eProDas_SendPeriodicPacket_AdvanceMany_FillBuffersBeforeStart
   (ByVal DeviceIdx As UInteger, ByRef Data As Byte) As Integer
```

As the name of the function suggests, internally the function `SendPeriodicPacket_AdvanceMany` is utilized for the task, so the pointer `Data` is automatically advanced properly to the next packet the in buffer. Make sure that the buffer contains at least twice as many packets as there are OUT buffer.

**Note 1.** You can initiate any data exchange with the eProDas device only after the function `PrepareForPeriodicActions` is successfully executed. Before that the periodic transfer engine is not configured and tuned yet.

**Note 2.** You must not use this function if you are using constant OUT buffer instead of true OUT packets (section 19.6.4).

### 19.10.2 FillConstantBufferBeforeStart

Prototypes in C/C++

```
int eProDas_FillConstBufferBeforeStart(unsigned int DeviceIdx,
  unsigned char *Data);
```

Prototypes in Delphi

```
function eProDas_FillConstBufferBeforeStart(DeviceIdx: LongWord;
  Data: PByte):integer;
```

Prototypes in VisualBasic

```
Function eProDas_FillConstBufferBeforeStart (ByVal DeviceIdx As UInteger,
  ByRef Data As Byte) As Integer
```

To fill in the constant OUT buffer, use function `FillConstBufferBeforeStart`. You do not specify the length of the buffer, since the function checks your selection from the previous configuration steps.

When constant OUT buffer is used, you must call this function before starting the race with the function `StartPeriodicActions`. Again, you can do it only after `PrepareForPeriodicActions` is done successfully.

### 19.10.3 Submitting initial receive requests

Okay, initial OUT packets or constant OUT buffer are under the roof. Now it is recommended to make pre-requests for IN packets so that the device driver will be able to pull them out of the device as soon as they are ready for the transmission. Here is the boilerplate code for the task (as a continuation of the Figure 92).

```
for(int i=0; I < PeriodicParams.NumberOf_IN_Buffers; i++) {
  eProDas_ReceivePeriodicPacket(DeviceIdx, AddressOf_IN_PacketWithIndex(i));
}
```

**Figure 93: Boilerplate code for demanding initial IN requests.**

This time we demand only as much IN transfers as there are buffers on the device and not the double amount. The reason is (hopefully) obvious: the device driver cannot pre-receive the packets, which are not formed yet, so it can only accept `NumberOf_IN_Buffers` requests. Instead of coding the boring fragment in the Figure 93 we again prepared a special function for the task.

### 19.10.3.1 ReceivePeriodicPacket_AdvanceMany_FillBuffersBeforeStart

Prototypes in C/C++

```
int ePro Das_ReceivePeriodicPacket_AdvanceMany_FillBuffersBeforeStart
  (unsigned int DeviceIdx, unsigned char *&Data);
```

Prototypes in Delphi

```
function eProDas_ReceivePeriodicPacket_AdvanceMany_FillBuffersBeforeStart
  (DeviceIdx: LongWord; var Data: PByte):integer;
```

Prototypes in VisualBasic

```
Function eProDas_ReceivePeriodicPacket_AdvanceMany_FillBuffersBeforeStart
  (ByVal DeviceIdx As UInteger, ByRef Data As Byte) As Integer
```

As the name of the function reveals, internally the function `ReceivePeriodicPacket_AdvanceMany` is utilized for the task, so the pointer `Data` is automatically advanced properly to the next packet's placeholder in buffer. Make sure that the buffer contains at least the room for as many packets as there are physical IN buffers on the device.

## 19.11 Stopping periodic program decently

The purpose of everything that we have been discussing so far in the chapter is to program and configure the eProDas device properly to execute periodic actions according to your preferences. Now we focus our attention to the last missing piece in the mosaic: how to stop periodic program decently.

We have already stressed that the application must provide OUT packets with excitation data on a regular basis during the running and it must also pull the generated data that are being sent through IN packets from the device at a sufficient pace. Now imagine that at this moment your application runs periodic actions and it needs three additional IN packets to be received and then the measurements are done. You simply call any suitable function for demanding IN transfer for three more times and then you decide to stop periodic actions.

The problem is that your application regains control immediately upon submission of the final IN requests although results of measurements, which are going to be stuffed into these IN packets, are yet to be obtained. If the frequency of periodic actions is extremely low it is possible that the outstanding measurements will be done only after several hours or days pass by. But if you stop periodic actions immediately, you cannot collect these missing and valuable results.

Of course, we could simply say that it is responsibility of your application to wait till everything is set and done before terminating periodic activities. Fortunately, the already mentioned function `StopPeriodicActions` (section 19.3.11.1) takes this burden onto itself. Whenever you initiate transmission or reception of any kind of packet eProDas device stack takes notes of that. The function `StopPeriodicActions` exploits this knowledge to decently complete all your outstanding requests before periodic actions are truly stopped. This happens only if you demand it to do so; alternatively you may abort the whole thing instantly.

Let us refresh the definition of the function `StopPeriodicActions`, which you have already seen.

### 19.11.1.1 StopPeriodicActions (repeated definition)

Prototype in C/C++

```
int eProDas_StopPeriodicActions(unsigned int DeviceIdx, unsigned int FinishSent,
  unsigned int FinishReceived, unsigned char *PostSendBuffer,
  unsigned int &SendViolations, unsigned int &ReceiveViolations);
```

Prototype in Delphi

```
function eProDas_StopPeriodicActions(DeviceIdx: LongWord; FinishSent: LongWord;
  FinishReceived: LongWord; PostOUTBuffervar: PByte; var SendViolations: LongWord;
  var ReceiveViolations: LongWord): integer;
```

Prototype in VisualBasic

```
Function eProDas_StopPeriodicActions (ByVal DeviceIdx As UInteger,
  ByVal FinishSent As UInteger, ByVal FinishReceived As UInteger,
  ByRef PostOUTBuffervar As Byte, ByRef SendViolations As UInteger,
  ByRef ReceiveViolations As UInteger) As Integer
```

A non-zero value of the parameter `FinishSent` instructs eProDas to complete all outstanding OUT transactions before it actually stops the program. Similarly, the non-zero parameter `FinishReceived` demands that all outstanding IN packets are completed before the program stops. If both of these parameters are zero, eProDas immediately aborts the execution of periodic actions regardless of the current state of transactions.

**Note.** The virtue of this function is its extreme patience. If, for example, you run your periodic program at clock speed of one execution per day and you demand 10 additional measurements (executions of periodic program), then the function `StopPeriodicActions` will faithfully wait for the next 10 days to collect the measurements and it will return to you only after this time span elapses.

Such behaviour is well suitable for running periodic actions at high frequencies where additional measurements take only a fraction of a second to complete. In such cases the described feature enables you to write much simpler code since you can omit all checking and waiting for completion of the demanded measurements. However, for slowly executing periodic actions you must decide for yourself whether this strict bookkeeping is beneficiary or do you prefer to abort the thing altogether.

### 19.11.1.2 Finding out about any potential violations

When the activities finally stop function `StopPeriodicActions` queries the device to discover whether there were any buffer underruns (timing violations on OUT packets) and/or buffer overruns (timing violations on IN packets) during the periodic execution. Both of these phenomena are due to the fact that OUT and/or IN packets were not delivered on time. It could be that your application does not receive/send them fast enough or you simply demand more from the USB connection than is technologically feasible. Slow frequency of execution down a bit and try again.

Buffer underruns and buffer overruns are reported as a non-zero returned values of parameters `SendViolations` and `ReceiveViolations`, respectively. Internally, eProDas device keeps a separate 16-bit counter for each of the event. The counter `SendViolations` (`ReceiveViolations`) is increased every time that buffer underrun (overrun) occurs. If violations are so excessive that a 16-bit counter is forced to roll over from 65,536 to 0, an additional counter-overflow indicator is set. This indicator is appended to the final result as a seventieth bit of the counter.

Therefore, if some violation counter holds a number between 1 and 65,535 this means that this is an exact number of encountered violations. However, if the number is greater than 65,536, this means that the base 16-bit counter has rolled over at least once. The reported number of violations may be exact or too little; in the case of a second rollover the reported number is too low for 65,536 violations; in the case of a third rollover, the reported number is too low for 2 x 65,536, etc.

Further, if only one occurrence of data corruption is detected during periodic run, both indicators of violations are set to the maximal possible 32-bit unsigned integer value (0xFFFFFFFF).

### 19.11.1.3 More about completion of outstanding transactions

Let us go back to the previous topic of a decent completion of outstanding transactions. When you decide not to abort the activities things begin to complicate (for the function, not for you). eProDas device is designed in such a way that it executes periodic program as fast as it can. That is why it needs all those previously described analyses and preparations: to assure that all internally utilized data structures are in place and initialized properly so that no error checking or other overhead introducing activities needs to happen during execution. If everything is not prepared well the device would simply crash or deadlock during its hard work.

Another consequence of the previous paragraph is that the device does not possess any code to distinguish between normal execution of periodic program and waiting for its completion since inclusion of such code would slow down things a bit. Therefore, until all data is generated and pulled out of chip the function `StopPeriodicActions` must make eProDas device believe that periodic actions execute normally and for that matter it has to continue transferring phantom OUT and IN packets until all your outstanding demands are fulfilled. Failing to do so may unnecessarily trigger `ReceiveViolations` and `SendViolations` indicators, which has to be avoided or you will have no clue about whether the reported violations are due to improper completion process or there were true problems during periodic execution. In other words, these two indicators would become useless.

Handling of IN packets is easy. The function simply takes care that there are always outstanding (phantom) demands for IN packets. If the device delivers something that the user did not request it is simply thrown into to the well without the bottom. OUT packets are more difficult to deal with, however, since they carry the data that need to be written to digital ports and to other connected circuits. Depending on the situation improper excitation values may prevent your setup to work properly during the transition from normal operation to the stopped steady state.

For that matter the function `StopPeriodicActions` allows you to specify a pointer `PostOUTBuffer` to a pre-initialized packet buffer that contains data to be sent to the eProDas device as phantom OUT packets until periodic actions are truly finished. If you do not care about the contents of these OUT packets specify zero value for this pointer and the function `StopPeriodicActions` will provide its own zero-filled buffer, which should be okay for the general case.

## 19.12 Periodic instructions

Now that we know everything about configuration and running of periodic actions the time has come to get detailed information about periodic instructions that are available to us. Names of all of them begin with a literal `eProDas_AddPeriodicAction_` and each one of them adds one action to the list of actions that are executed during each period. The actions should be familiar already, since they enable you to do the same things as the ordinary function do, except that now the efficiency is increased and instants in time when certain action executes are defined much more precisely.

### 19.12.1 Working with digital ports (part one: reading and writing data)

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_ReadPort(unsigned int DeviceIdx, unsigned int PortIdx);

int eProDas_AddPeriodicAction_ReadLatch(unsigned int DeviceIdx, unsigned int PortIdx);

int eProDas_AddPeriodicAction_ReadTris(unsigned int DeviceIdx, unsigned int PortIdx);

int eProDas_AddPeriodicAction_WritePort(unsigned int DeviceIdx, unsigned int PortIdx);

int eProDas_AddPeriodicAction_WriteTris(unsigned int DeviceIdx, unsigned int PortIdx);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_ReadPort(DeviceIdx: LongWord; PortIdx: LongWord):integer;

function eProDas_AddPeriodicAction_ReadLatch(DeviceIdx: LongWord; PortIdx: LongWord):integer;

function eProDas_AddPeriodicAction_ReadTris(DeviceIdx: LongWord; PortIdx: LongWord):integer;

function eProDas_AddPeriodicAction_WritePort(DeviceIdx: LongWord; PortIdx: LongWord):integer;

function eProDas_AddPeriodicAction_WriteTris(DeviceIdx: LongWord; PortIdx: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_AddPeriodicAction_ReadPort (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger) As Integer

Function eProDas_AddPeriodicAction_ReadLatch (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger) As Integer

Function eProDas_AddPeriodicAction_ReadTris (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger) As Integer

Function eProDas_AddPeriodicAction_WritePort (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger) As Integer

Function eProDas_AddPeriodicAction_WriteTris (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger) As Integer
```

Parameter `PortIdx` specifies port on which a certain action is done (Table 4 on page 67). Action `ReadPort` reads the current state of the port (its pins) and writes the result into the IN pipe. `ReadLatch` reads port's latch register instead of pins'. Finally, `ReadTris` obtains the value of port's TRIS register.

Action `WritePort` reads a byte from OUT pipe and writes it to the port, whereas in the case of action `WriteTris` the destination is port's TRIS register. The former function enables you to implement arbitrary excitation, whereas the latter enables you to reconfigure port in an arbitrary way.

The following table summarizes some characteristics of these actions. The first column lists abbreviated command names, where the literal `eProDas_AddPeriodicAction_` has been dropped. The second column specifies durations in PIC's instruction cycles (ticks of 12 MHz clock or multiples of 83.3 ns). These numbers are also reported by the periodic analyzer.

If command sends any data like results of measurements to the host PC the third column specifies the number of bytes that are added to the IN pipe. Similarly, if command consumes some data from the OUT pipe the fourth column reveals the number of consumed bytes.

| command name | duration (ICs) | USB IN (bytes) | USB OUT (bytes) |
|---|---|---|---|
| `ReadPort` | 2 | 1 | |
| `ReadLatch` | 2 | 1 | |
| `ReadTris` | 2 | 1 | |
| `WritePort` | 2 | | 1 |
| `WriteTris` | 2 | | 1 |

**Table 13: Summary of actions on digital ports (part one).**

## 19.12.2 Working with digital ports (part two: constants, clear, set and complement)

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_WriteConst2Port(unsigned int DeviceIdx, unsigned int PortIdx,
  unsigned int Const);

int eProDas_AddPeriodicAction_WriteConst2Tris(unsigned int DeviceIdx, unsigned int PortIdx,
  unsigned int Const);

int eProDas_AddPeriodicAction_ClearPort(unsigned int DeviceIdx, unsigned int PortIdx);

int eProDas_AddPeriodicAction_SetPort(unsigned int DeviceIdx, unsigned int PortIdx);

int eProDas_AddPeriodicAction_ComplementPort(unsigned int DeviceIdx, unsigned int PortIdx);

int eProDas_AddPeriodicAction_ClearPin(unsigned int DeviceIdx,
  unsigned int PortIdx, unsigned int PinIdx);

int eProDas_AddPeriodicAction_SetPin(unsigned int DeviceIdx,
  unsigned int PortIdx, unsigned int PinIdx);

int eProDas_AddPeriodicAction_TogglePin(unsigned int DeviceIdx,
  unsigned int PortIdx, unsigned int PinIdx);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_WriteConst2Port(DeviceIdx: LongWord; PortIdx: LongWord;
  Constant: LongWord):integer;

function eProDas_AddPeriodicAction_WriteConst2Tris(DeviceIdx: LongWord; PortIdx: LongWord;
  Constant: LongWord):integer;

function eProDas_AddPeriodicAction_ClearPort(DeviceIdx: LongWord; PortIdx: LongWord):integer;

function eProDas_AddPeriodicAction_SetPort(DeviceIdx: LongWord; PortIdx: LongWord):integer;

function eProDas_AddPeriodicAction_ComplementPort(DeviceIdx: LongWord;
  PortIdx: LongWord):integer;

function eProDas_AddPeriodicAction_ClearPin(DeviceIdx: LongWord;
  PortIdx: LongWord; PinIdx: LongWord):integer;

function eProDas_AddPeriodicAction_SetPin(DeviceIdx: LongWord;
  PortIdx: LongWord; PinIdx: LongWord):integer;

function eProDas_AddPeriodicAction_TogglePin(DeviceIdx: LongWord;
  PortIdx: LongWord; PinIdx: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_AddPeriodicAction_WriteConst2Port (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger, ByVal Constant As UInteger) As Integer

Function eProDas_AddPeriodicAction_WriteConst2Tris (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger, ByVal Constant As UInteger) As Integer

Function eProDas_AddPeriodicAction_ClearPort (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger) As Integer

Function eProDas_AddPeriodicAction_SetPort (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger) As Integer

Function eProDas_AddPeriodicAction_ComplementPort (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger) As Integer

Function eProDas_AddPeriodicAction_ClearPin (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger, ByVal PinIdx As UInteger) As Integer

Function eProDas_AddPeriodicAction_SetPin (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger, ByVal PinIdx As UInteger) As Integer

Function eProDas_AddPeriodicAction_TogglePin (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger, ByVal PinIdx As UInteger) As Integer
```

To write the constant to a port (latch), use the action `WriteConst2Port` (`WriteConst2Tris`). The action `WriteConst2Port` is handy for generation of e.g. bus control signals to simulate microprocessor's read/write cycles or something similar; the action `WriteConst2Tris` enables us to reconfigure port in an arbitrary (but constant) way during bus cycles. These two actions are attractive since they do not consume OUT pipe, so they spare USB packets' contents for real data.

Use actions `ClearPort` and `SetPort` to clear or set all pins of a port, respectively. To complement all pins, use the action `ComplementPort`. The latter may not be of much use, but the first two may be handy when some port is uses as a generator of control signals, which you all want to put into a disabled state. No USB pipes are consumed for these tasks, which makes these functions attractive.

There are also actions that perform these operations on an isolated pin, which is what you need in the majority of cases. Actions `ClearPin` and `SetPin` put certain pin into low or high state, respectively, whereas `TogglePin` toggles its current state. Pin within port is specified through the parameter `PinIdx` in a usual way. Again, no USB pipes are consumed by these actions, so you can implement microprocessor control signals without decreasing a scare USB bandwidth.

Summary of these commands is given in the following table. As you can see some of these actions consume only one PIC's instruction to execute, so you cannot go faster than that.

| command name | duration (ICs) | USB IN (bytes) | USB OUT (bytes) |
|---|---|---|---|
| `WriteConst2Port` | 2 | | |
| `WriteConst2Tris` | 2 | | |
| `ClearPort` | 1 | | |
| `SetPort` | 1 | | |
| `ComplementPort` | 1 | | |
| `ClearPin` | 1 | | |
| `SetPin` | 1 | | |
| `ComplementPin` | 1 | | |

**Table 14: Summary of actions on digital ports (part two).**

### 19.12.3 Working with digital ports (part three: direction of entire port)

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_ConfigPortAsOutput(unsigned int DeviceIdx,
  unsigned int PortIdx);

int eProDas_AddPeriodicAction_ConfigPortAsInput(unsigned int DeviceIdx, unsigned int PortIdx);

int eProDas_AddPeriodicAction_TogglePortDirection(unsigned int DeviceIdx,
  unsigned int PortIdx);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_ConfigPortAsOutput(DeviceIdx: LongWord;
  PortIdx: LongWord):integer;

function eProDas_AddPeriodicAction_ConfigPortAsInput(DeviceIdx: LongWord;
  PortIdx: LongWord):integer;

function eProDas_AddPeriodicAction_TogglePortDirection(DeviceIdx: LongWord;
  PortIdx: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_AddPeriodicAction_ConfigPortAsOutput (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger) As Integer

Function eProDas_AddPeriodicAction_ConfigPortAsInput (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger) As Integer

Function eProDas_AddPeriodicAction_TogglePortDirection (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger) As Integer
```

Use the actions `ConfigPortAsOutput` or `ConfigPortAsInput` to configure entire port as output or input, respectively. To change direction of all pins, use the action `TogglePortDirection`. None of these commands consume and USB pipe as the following table reveals.

| command name | duration (ICs) | USB IN (bytes) | USB OUT (bytes) |
|---|---|---|---|
| `ConfigPortAsOutput` | 1 | | |
| `ConfigPortAsInput` | 1 | | |
| `TogglePortDirection` | 1 | | |

**Table 15: Summary of actions on digital ports (part three).**

A good reason for reconfiguring the entire port(s) it to implement data buses of various widths. In our opinion the most likely combinations of ports for the task are the ones listed in the Table 16.

| port(s) combination | remarks |
|---|---|
| `D` | 8-bit data bus |
| `B` | 8-bit data bus |
| `D+E` | possibility of 10- or 12-bit data bus for reading, 10- or 11-bit data bus for writing |
| `B+E` | possibility of 10- or 12-bit data bus for reading, 10- or 11-bit data bus for writing |
| `D+B` | possibility of 16-bit wide data bus in both directions |
| `D+B+E` | possibility of 20-bit data bus for reading, 19-bit data bus for writing |
| `D+B+A` | possibility of 22-bit wide data bus in both directions |
| `D+B+A+E` | possibility of 26-bit data bus for reading, 25-bit data bus for writing |
| `D+B+A+E+C` | possibility of 31-bit data bus for reading, 30-bit data bus for writing |

**Table 16: Reading or writing more digital ports at a time (partial set).**

Port D is the most suitable port for implementing 8-bit wide data bus since it contains all 8 pins and these pins do not implement much other functionality, like being inputs of AD converter that you want to use in parallel with the data bus. For wider buses you combine this port with other one(s) in any way that you prefer but we cannot resist the temptation to give you some hints.

Sometimes you work with 10-bit or 12-bit external AD and DA converters with parallel interface for which you need a data bus of the appropriate size. Both, 10- and 12-bit options can be covered by combining port D and port E. Note that in this scenario port D should be read/written first and port E after that. This way we mimic little-endian bus architecture which prevails in the today's world; in order not to have bit gaps in the result port D must contain 8 the least significant bits of the data.

Also note that pin RE3 cannot be digital output so you can gain at most 11-bit bus for writing although you can implement 12-bit bus for reading by using the specified combination. Fortunately, there are many applications with more demand on precision of AD than DA; i.e. control system cannot control the controlled variable more precisely than this very variable is measured.

If the arrangement of your circuit demands that port D is occupied for some other task you can achieve the same functionality by combining port E with port B which is another 8-bit wide port.

For data buses of up to 16-bit width there is a handy combination of ports D and ports B.

Now we simply extend the concept as far as possible. By combining all three so far mentioned ports we gain the possibility of implementing 20-bit bus for reading and 19-bit bus for writing. Port E must carry the most significant bits if you use little-endian architecture; otherwise there are gaps in the bit pattern.

If you are willing to give away the analog functionality of port A you can achieve 22-bit data bus in both directions by combining ports D, B and A. You are probably wondering why anyone would use 22-bit wide bus and the truth is that we cannot come up with many examples. Still, if you are working with 24-bit sigma-delta AD converter and you know that noise in the analog path blurs a couple of LSB bits you might not care to lose two the least significant ones. (On the other hand these AD converters usually posses some sort of serial interface, so the argument is not particularly realistic ☹.)

As it turns out 22-bits are the most that we can squeeze out of PIC without holes in bit sequence. If we can live without bits being completely stuffed together still wider buses are possible to implement. By adding port E to the previous combinations we can get 26-bits for reading and 25-bits for writing. Now you can at least implement a true 24-bit bus although you need to take care of the gap in the bit pattern due to non-available or missing pins RA6 and RA7.

And finally by combining all five PIC's ports into one bus we can achieve 31-bit bus for reading and 30-bit bus for writing. Unfortunately, one pin is missing to implement 32-bit data exchange between PIC and its digital surrounding. In addition, there is no room for any control signals, so the question is whether this option is of any practical value. Well, we are sure that your imagination cannot be hindered by such small obstacles. ☺

### 19.12.4 Working with digital ports (part four: direction of isolated pins)

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_ConfigPinAsOutput(unsigned int DeviceIdx, unsigned int PortIdx,
  unsigned int PinIdx);

int eProDas_AddPeriodicAction_ConfigPinAsInput(unsigned int DeviceIdx, unsigned int PortIdx,
  unsigned int PinIdx);

int eProDas_AddPeriodicAction_TogglePinDirection(unsigned int DeviceIdx, unsigned int PortIdx,
  unsigned int PinIdx);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_ConfigPinAsOutput(DeviceIdx: LongWord;
  PortIdx: LongWord; PinIdx: LongWord):integer;

function eProDas_AddPeriodicAction_ConfigPinAsInput(DeviceIdx: LongWord;
  PortIdx: LongWord; PinIdx: LongWord):integer;

function eProDas_AddPeriodicAction_TogglePinDirection(DeviceIdx: LongWord;
  PortIdx: LongWord; PinIdx: LongWord):integer;
```

Prototypes in VisualBasic

```
eProDas_AddPeriodicAction_ConfigPinAsOutput (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger, ByVal PinIdx As UInteger) As Integer

Function eProDas_AddPeriodicAction_ConfigPinAsInput (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger, ByVal PinIdx As UInteger) As Integer

Function eProDas_AddPeriodicAction_TogglePinDirection (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger, ByVal PinIdx As UInteger) As Integer
```

Instead of changing direction of the entire port, now we change only direction of isolated pins, which is handy in many situations. Summary of these functions is as follows.

| command name | duration (ICs) | USB IN (bytes) | USB OUT (bytes) |
|---|---|---|---|
| ConfigPinOutput | 1 | | |
| ConfigPinAsInput | 1 | | |
| TogglePinDirection | 1 | | |

**Table 17: Summary of actions on digital ports (part four).**

### 19.12.5 Working with digital ports (part five: value and direction of selected pins)

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_AND_ConstWithPort(unsigned int DeviceIdx,
  unsigned int PortIdx, unsigned int Const);

int eProDas_AddPeriodicAction_OR_ConstWithPort(unsigned int DeviceIdx,
  unsigned int PortIdx, unsigned int Const);

int eProDas_AddPeriodicAction_XOR_ConstWithPort(unsigned int DeviceIdx,
  unsigned int PortIdx, unsigned int Const);

int eProDas_AddPeriodicAction_AND_ConstWithTris(unsigned int DeviceIdx,
  unsigned int PortIdx, unsigned int Const);

int eProDas_AddPeriodicAction_OR_ConstWithTris(unsigned int DeviceIdx,
  unsigned int PortIdx, unsigned int Const);

int eProDas_AddPeriodicAction_XOR_ConstWithTris(unsigned int DeviceIdx,
  unsigned int PortIdx, unsigned int Const);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_AND_ConstWithPort(DeviceIdx: LongWord; PortIdx: LongWord;
  Constant: LongWord):integer;

function eProDas_AddPeriodicAction_OR_ConstWithPort(DeviceIdx: LongWord; PortIdx: LongWord;
  Constant: LongWord):integer;

function eProDas_AddPeriodicAction_XOR_ConstWithPort(DeviceIdx: LongWord; PortIdx: LongWord;
  Constant: LongWord):integer;

function eProDas_AddPeriodicAction_AND_ConstWithTris(DeviceIdx: LongWord; PortIdx: LongWord;
  Constant: LongWord):integer;

function eProDas_AddPeriodicAction_OR_ConstWithTris(DeviceIdx: LongWord; PortIdx: LongWord;
  Constant: LongWord):integer;

function eProDas_AddPeriodicAction_XOR_ConstWithTris(DeviceIdx: LongWord; PortIdx: LongWord;
  Constant: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_AddPeriodicAction_AND_ConstWithPort (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger, ByVal Constant As UInteger) As Integer

Function eProDas_AddPeriodicAction_OR_ConstWithPort (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger, ByVal Constant As UInteger) As Integer

Function eProDas_AddPeriodicAction_XOR_ConstWithPort (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger, ByVal Constant As UInteger) As Integer

Function eProDas_AddPeriodicAction_AND_ConstWithTris (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger, ByVal Constant As UInteger) As Integer

Function eProDas_AddPeriodicAction_OR_ConstWithTris (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger, ByVal Constant As UInteger) As Integer

Function eProDas_AddPeriodicAction_XOR_ConstWithTris (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger, ByVal Constant As UInteger) As Integer
```

With this set of actions you can do bitwise AND, OR and XOR between constant of your choice and any port or its TRIS. All these functions consume 2 ICs of execution time and they do not touch any of the USB pipes.

### 19.12.6 Working with digital ports (part six: address busses)

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_IncrementPort(unsigned int DeviceIdx, unsigned int PortIdx);

int eProDas_AddPeriodicAction_IncrementSubPort(unsigned int DeviceIdx, unsigned int PortIdx,
  unsigned int StartPin, unsigned int EndPin);

int eProDas_AddPeriodicAction_IncrementSubPortWithReset(unsigned int DeviceIdx,
  unsigned int PortIdx, unsigned int StartPin, unsigned int EndPin, unsigned int MatchValue,
  unsigned int ResetValue);

int eProDas_AddPeriodicAction_RotateLeftPort(unsigned int DeviceIdx, unsigned int PortIdx);

int eProDas_AddPeriodicAction_RotateLeftSubPort(unsigned int DeviceIdx, unsigned int PortIdx,
  unsigned int StartPin, unsigned int EndPin);

int eProDas_AddPeriodicAction_RotateLeftSubPortWithReset(unsigned int DeviceIdx,
  unsigned int PortIdx, unsigned int StartPin, unsigned int EndPin, unsigned int MatchValue,
  unsigned int ResetValue);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_IncrementPort(DeviceIdx: LongWord;
  PortIdx: LongWord):integer;

function eProDas_AddPeriodicAction_IncrementSubPort(DeviceIdx: LongWord; PortIdx: LongWord;
  StartPin: LongWord; EndPin: LongWord):integer;

function eProDas_AddPeriodicAction_IncrementSubPortWithReset(DeviceIdx: LongWord;
  PortIdx: LongWord; StartPin: LongWord; EndPin: LongWord; MatchValue: LongWord;
  ResetValue: LongWord):integer;

function eProDas_AddPeriodicAction_RotateLeftPort(DeviceIdx: LongWord;
  PortIdx: LongWord):integer;

function eProDas_AddPeriodicAction_RotateLeftSubPort(DeviceIdx: LongWord; PortIdx: LongWord;
  StartPin: LongWord; EndPin: LongWord):integer;

function eProDas_AddPeriodicAction_RotateLeftSubPortWithReset(DeviceIdx: LongWord;
  PortIdx: LongWord; StartPin: LongWord; EndPin: LongWord; MatchValue: LongWord;
  ResetValue: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_AddPeriodicAction_IncrementPort (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger) As Integer

Function eProDas_AddPeriodicAction_IncrementSubPort (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger, ByVal StartPin As UInteger, ByVal EndPin As UInteger) As Integer

Function eProDas_AddPeriodicAction_IncrementSubPortWithReset (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger, ByVal StartPin As UInteger, ByVal EndPin As UInteger,
  ByVal MatchValue As UInteger, ByVal ResetValue As UInteger) As Integer

Function eProDas_AddPeriodicAction_RotateLeftPort (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger) As Integer

Function eProDas_AddPeriodicAction_RotateLeftSubPort (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger, ByVal StartPin As UInteger, ByVal EndPin As UInteger) As Integer

Function eProDas_AddPeriodicAction_RotateLeftSubPortWithReset (ByVal DeviceIdx As UInteger,
  ByVal PortIdx As UInteger, ByVal StartPin As UInteger, ByVal EndPin As UInteger,
  ByVal MatchValue As UInteger, ByVal ResetValue As UInteger) As Integer
```

With this set of self descriptive actions you can implement incrementing as well as rotating address bus schemes that we were discussing back in the section 16.2. Where appropriate `StartPin` and `EndPin` specify a subset of adjacent pins on which the action is taken, whereas `MatchValue` specify a matching condition upon which the port's value is reset to `ResetValue`. These actions do not touch USB pipes. The execution times vary significantly among the functions, as the following table shows.

| command name | duration (ICs) | USB IN (bytes) | USB OUT (bytes) |
|---|---|---|---|
| IncrementPort | 1 | | |
| IncrementSubPort | 8 | | |
| IncrementSupPortWithReset | 12 | | |
| RotateLeftPort | 1 | | |
| RotateLeftSubPort | 10 | | |
| RotateLeftSubPortWithReset | 14 | | |

**Table 18: Summary of actions on digital ports (part six).**

### 19.12.7 Actions on internal AD and comparators

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_ReadInternalAD(unsigned int DeviceIdx);

int eProDas_AddPeriodicAction_ReadInternalAD_8BitLeftAligned(unsigned int DeviceIdx);

int eProDas_AddPeriodicAction_ReadInternalComparators(unsigned int DeviceIdx);

int eProDas_AddPeriodicAction_ReadInternalAD_Comparators(unsigned int DeviceIdx);

int eProDas_AddPeriodicAction_WaitInternalAD(unsigned int DeviceIdx);

int eProDas_AddPeriodicAction_StartInternalAD(unsigned int DeviceIdx);

int eProDas_AddPeriodicAction_AcquireConstInternalAD(unsigned int DeviceIdx,
  unsigned int ADChannel);

int eProDas_AddPeriodicAction_AcquireInternalAD(unsigned int DeviceIdx);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_ReadInternalAD(DeviceIdx: LongWord):integer;

function eProDas_AddPeriodicAction_ReadInternalAD_8BitLeftAligned(
DeviceIdx: LongWord):integer;

function eProDas_AddPeriodicAction_ReadInternalComparators(DeviceIdx: LongWord):integer;

function eProDas_AddPeriodicAction_ReadInternalAD_Comparators(DeviceIdx: LongWord):integer;

function eProDas_AddPeriodicAction_WaitInternalAD(DeviceIdx: LongWord):integer;

function eProDas_AddPeriodicAction_StartInternalAD(DeviceIdx: LongWord):integer;

function eProDas_AddPeriodicAction_AcquireConstInternalAD(DeviceIdx: LongWord;
  ADChannel: LongWord):integer;

function eProDas_AddPeriodicAction_AcquireInternalAD(DeviceIdx: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_AddPeriodicAction_ReadInternalAD (ByVal DeviceIdx As UInteger) As Integer

Function eProDas_AddPeriodicAction_ReadInternalAD_8BitLeftAligned (
  ByVal DeviceIdx As UInteger) As Integer

Function eProDas_AddPeriodicAction_ReadInternalComparators (ByVal DeviceIdx As UInteger)
  As Integer

Function eProDas_AddPeriodicAction_ReadInternalAD_Comparators (ByVal DeviceIdx As UInteger)
  As Integer

Function eProDas_AddPeriodicAction_WaitInternalAD (ByVal DeviceIdx As UInteger) As Integer

Function eProDas_AddPeriodicAction_StartInternalAD (ByVal DeviceIdx As UInteger) As Integer
  ByVal PortIdx As UInteger) As Integer

Function eProDas_AddPeriodicAction_AcquireConstInternalAD (ByVal DeviceIdx As UInteger,
  ByVal ADChannel As UInteger) As Integer

Function eProDas_AddPeriodicAction_AcquireInternalAD (ByVal DeviceIdx As UInteger) As Integer
```

Action `ReadInternalAD` obtains the result of the last **completed** AD conversion and writes it into the IN pipe; the low byte of the result is written first and the high byte after that (results are 10-bits wide). Note that it is not an error to consume the result from the previous period of program execution if things are properly organized. Simply, this action grabs the last AD result that AD converter produced and it does not care about when the result got ready.

In the cases where USB bandwidth is more important than the precision of AD result, you may obtain only the upper 8-bits of the result, so only one byte of IN pipe is consumed instead of two. The action for the task is `ReadInternalAD_8BitLeftAligned`. For this to work, AD converter must be configured to deliver the so-called left aligned results (section 15.2.2).

Action `ReadInternalComparators` reads the current state of internal comparators and writes it to the IN pipe. The state of comparators 1 and 2 are contained in the 6th and 7th bit of this byte, respectively.

Action `ReadInternalAD_Comparators` reads the current AD result and the state of comparators together, by means of which one byte of IN pipe is spared in comparison to the execution of these two actions separately. First, the low byte of AD result is written to the IN pipe, next the state of comparators (bits 6 and 7) is glued together with the two MSB bits of AD result (bits 0 and 1) and written to the IN pipe. Your application must properly decompose the second byte (presumably by using bit oriented operations).

Action `WaitInternalAD` waits that the currently in progress AD conversion is completed. This is useful e.g. if your periodic program has already done all other activities but the AD result is still not ready. If AD conversion is not in progress, this action simply consumes 3 ICs of PIC's processor time, without doing any other harm.

Internal AD converter is started automatically at the beginning of the period. When AD result is consumed, you may want to start another AD conversion (perhaps your want to change the AD channel first; see below) to deliver several AD results in one period of execution. To start another conversion, use add action `StartInternalAD` to your periodic program.

### 19.12.7.1 How to select AD channel for conversion

If you intend to sample only one AD channel during periodic actions (i.e. the same one during all repetitions of the periodic program) simply acquire this particular channel before configuring periodic actions, for example by means of command `eProDas_AcquireInternalADChannel` (section 15.2.3). After AD conversion is over the channel does not change by itself and the next AD conversion targets the same channel over and over again.

To change the channel during periodic actions use the action `AcquireConstInternalAD`. The channel of your choice is specified by parameter `ADChannel`. Note that when you change the channel the acquisition period starts and you may need to wait certain time before starting the AD conversion to obtain accurate results (section 15.2.1, appendix B). To insert this time gap automatically when AD conversion is started, configure AD module accordingly (Table 7 on page 79).

The second situation arises when you intend to read more AD channels during one period. In such cases the best (i.e. the fastest) solution is to pre-acquire the first channel before configuration of periodic actions by **non-periodic** command `eProDas_AcquireInternalADChannel` (section 15.2.3). Then, when the first result is consumed by action `ReadInternalAD`, use **periodic** action `AcquireInternalADChannel` (from this section) to acquire the next intended channel. Also, before periodic actions exit (or better, as soon as the last AD conversion is over), use the action `AcquireInternalADChannel` one more time to select the first channel to be the target for the AD in next period (presumably, this would be the channel that you selected by non-periodic acquisition).

To gain the ultimate flexibility of AD channel selection, use the action `AcquireInternalAD`, which reads the requested channel number from OUT pipe. This way, you can specify different AD channel for each period of execution. Note however, that you cannot just write the raw channel number into the OUT packet, but you must use the following function to convert AD channel number into the proper value for the OUT packet.

#### 19.12.7.1.1CalcInternalAD_AcquisitionCode

Prototype in C/C++

```
int eProDas_AddPeriodicAction_CalcInternalAD_AcquisitionCode(
  unsigned int ADChannel, unsigned char &Code);
```

Prototype in Delphi

```
function eProDas_CalcInternalAD_AcquisitionCode(ADChannel: LongWord;
  var Code: Byte):integer;
```

Prototype in VisualBasic

```
Function eProDas_CalcInternalAD_AcquisitionCode (ByVal ADChannel As UInteger,
  ByRef Code As Byte) As Integer
```

If possible, arrange things in such way that you acquire AD channel at the end of periodic program (i.e. after the last AD conversion) and consume the result in the next period. This way the dead time between two adjacent periods can be exploited as an acquisition time. Although, periodic analyzer cannot cope with such scenarios and it may issue some errors or warnings, feel confident that you are doing the right thing.

| command name | duration (ICs) | USB IN (bytes) | USB OUT (bytes) |
|---|---|---|---|
| `ReadInternalAD` | 4 | 2 | |
| `ReadInternalAD_8BitLeftAligned` | 2 | 1 | |
| `ReadInternalComparators` | 2 | 1 | |
| `ReadInternalAD_Comparators` | 5 | 2 | |
| `WaitInternalAD` | max. 5 + ICs till the end of AD conversion | | |
| `StartInternalAD` | 1 | | |
| `AcquireConstInternalAD` | 2 | | |
| `AcquireInternalAD` | 2 | | 1 |

**Table 19: Summary of actions on internal AD and comparators.**

### 19.12.8 Actions on internal voltage reference

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_SetConstInternalVoltageReference(unsigned int DeviceIdx,
  unsigned int Enabled, unsigned int Value, unsigned int Range, unsigned int SourceOnPin,
  unsigned int OutputOnPin);

int eProDas_AddPeriodicAction_SetInternalVoltageReference(unsigned int DeviceIdx);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_SetConstInternalVoltageReference(DeviceIdx: LongWord;
  Enabled: LongWord; Value: LongWord; Range: LongWord; SourceOnPin: LongWord;
  OutputOnPin: LongWord):integer;

function eProDas_AddPeriodicAction_SetInternalVoltageReference(DeviceIdx: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_AddPeriodicAction_SetConstInternalVoltageReference (
  ByVal DeviceIdx As UInteger, ByVal Enabled As UInteger, ByVal Value As UInteger,
  ByVal Range As UInteger, ByVal SourceOnPin As UInteger, ByVal OutputOnPin As UInteger)
 As Integer

Function eProDas_AddPeriodicAction_SetInternalVoltageReference (ByVal DeviceIdx
  As UInteger) As Integer
```

Voltage reference is even simpler to deal with. You can only change the value of the output voltage, say, for altering trigger level of comparators or to exploit the feature as a simple 4-bit DA module. With the action `SetConstInternalVoltageReference` you specify full configuration of the module (see section 15.4.1 for explanation of parameters). Of course, such action is useful only if you change the reference several times during periodic actions. In the opposite case, simply configure the module before you start periodic actions in order not to waste PIC's time on reconfiguring the module with the same parameters over and over again.

For the utmost in flexibility, use the action `SetInternalVoltageReference`, which reads the configuration of the module from the OUT pipe, by means of which your application can alter the generated voltage in an arbitrary way during periodic actions. The proper value to be written to the OUT pipe by the application is calculated by the help of the following function.

#### 19.12.8.1.1 CalcVoltageReferenceCode

Prototype in C/C++

```
int eProDas_CalcVoltageReferenceCode(unsigned int Enabled, unsigned int Value,
  unsigned int Range, unsigned int SourceOnPin, unsigned int OutputOnPin,
  unsigned char &Code);
```

Prototype in Delphi

```
function eProDas_CalcVoltageReferenceCode(Enabled: LongWord; Value: LongWord;
  Range: LongWord; SourceOnPin: LongWord; OutputOnPin: LongWord;
  var Code: Byte):integer;
```

Prototype in VisualBasic

```
Function eProDas_CalcVoltageReferenceCode Lib "eProDas.dll" (
  ByVal Enabled As UInteger, ByVal Value As UInteger, ByVal Range As UInteger,
  ByVal SourceOnPin As UInteger, ByVal OutputOnPin As UInteger, ByRef Code As Byte)
  As Integer
```

**Note.** The output of the voltage reference is not amplified by any analog amplifier and therefore this voltage source has a rather large (Thevenin) internal resistance. The obvious consequence is that you must not load the output with any sort of low impedance circuitry without using voltage follower or some other impedance converter in-between. The less obvious consequence is that internal resistance forms a relatively large *RC* time constant with parasitic capacitances of pins, wires, connectors, etc. All this means that analog value of the voltage cannot change instantly and there can be adequately long time interval between the change of voltage reference configuration and the time when the new steady (i.e. expected) voltage is reached. Use your oscilloscope to see what is really happening and to test whether you are pushing the dynamics over the limits of the reality.

| command name | duration (ICs) | USB IN (bytes) | USB OUT (bytes) |
|---|---|---|---|
| SetConstInternalVoltageReference | 2 | | |
| SetInternalVoltageReference | 2 | | 1 |

**Table 20: Summary of actions on internal voltage reference.**

### 19.12.9 Adding delay to execution

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_Delay(unsigned int DeviceIdx, unsigned int Length_ICs);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_Delay(DeviceIdx: LongWord; Length_ICs: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_AddPeriodicAction_Delay (ByVal DeviceIdx As UInteger,
  ByVal Length_ICs As UInteger) As Integer
```

Sometimes you may encounter a situation where your externally hooked periphery cannot keep pace with the PIC. A typical example may be a slow LCD module, which requires bus cycle length of 300 ns or so. Since the majority of PIC's instructions are processed in 83.3 ns it is possible to write periodic program that generates too fast microprocessor bus cycles in such cases. To remedy the situation, insert small delay into your program with action `Delay`. The parameter `Length_ICs` specify desired duration length in number of PIC's instruction cycles; therefore, these delays are always an integer multiple of 83.3 ns.

Please note. This action is intended to be used in exactly the scenarios like the previously paragraph describes. For each requested `Length_ICs` eProDas adds one NOP (no-operation) PIC instruction to your periodic program, which means that specifying 1,000 for `Length_ICs` adds one thousand NOP instructions to your program. There are no program loops to synthesize long pauses.

### 19.12.10 Flushing PIC's USB transactions FIFO; a must read section

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_FlushUSB(unsigned int DeviceIdx);

int eProDas_AddPeriodicAction_FlushUSB_WaitInternalAD(unsigned int DeviceIdx);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_FlushUSB(DeviceIdx: LongWord):integer;

function eProDas_AddPeriodicAction_FlushUSB_WaitInternalAD(DeviceIdx: LongWord):integer;
```

Prototypes in VisualBasic

```
Function eProDas_AddPeriodicAction_FlushUSB (ByVal DeviceIdx As UInteger) As Integer

Function eProDas_AddPeriodicAction_FlushUSB_WaitInternalAD (ByVal DeviceIdx As UInteger)
  As Integer
```

Now we have to deal with an extremely technical and boring issue that we would like not to have the need to discuss at all. Please, be patient since failing to acknowledge the material in this section may severely hinder periodic actions in certain scenarios of running.

Internally, PIC possesses an USB transactions FIFO queue into which it logs all successful receptions and transmissions of USB packets. Whenever a new OUT packet arrives from the host PC or a new IN packet is delivered to the host PC, one entry is added to this USB transactions FIFO. The purpose of this logging is that the firmware on the chip can have an insight into completed USB transactions; whenever firmware acknowledges certain transaction it deletes one entry from the queue.

The problem is that this FIFO can hold only 4 entries and when it becomes full (firmware does not clean entries fast enough) PIC stops further transmissions and receptions of USB packets (for the familiar with the USB standard: PIC starts sending NAK responses to the host PC) until the queue gets emptied somehow.

Now, let us recall that eProDas periodic engine processes USB packets at the beginning and/or at the end of periodic program (section 19.4.1.1, Figure 70 on page 189). Further, in order to assure as jitterless operation as possible processing of USB transactions always takes the same amount of time whether there are any transactions to process or not. When the current OUT and/or IN buffer is not swapped with a new one, eProDas exploits the idle time to clean USB transactions FIFO queue, so generally you do not need to worry about the issue at all.

Also, one entry from the queue is deleted during the execution of entry routine (again, take a look a Figure 70 on page 189). This enables the device to receive `StopPeriodicActions` command even if no USB packets are utilized during periodic program.

Now imagine that your program uses OUT as well as IN packets but you instructed both of them to be delivered after each period of execution (by setting parameters of the function `SetPeriodicMode` to the appropriate values). Since both buffers are swapped during each period, eProDas does not have any idle time to clean USB transactions FIFO queue. One entry is deleted upon execution of the entry routine, but this is not enough since two packets (one OUT and one IN) are delivered during each period of execution.

Accordingly, after two periods of execution the FIFO queue becomes stuffed with transaction logs and PIC stops further USB transactions, which inevitably leads to violations. When violations are discovered, eProDas takes its time and cleans FIFO queue, so after a small delay the operations proceed as normal. Nonetheless, such behaviour should be avoided and for that matter you need to clean up the FIFO queue manually if you deliver both types of packets after each period of execution. The action `FlushUSB` cleans one record from the FIFO queue and this is the minimum that you should do in the described case.

Further, it is wise to clean more entries (up to four) if you have the time and such addition does not imposes bottleneck on periodic execution. The more the FIFO queue is emptied the smaller is the chance of violations.

You can even empty the queue during waiting on AD conversion by using the action `FlushUSB_WaitInternalAD` instead of plain `WaitInternalAD`. This action flushes one entry from FIFO queue and checks whether AD conversion is over. If not, the activity repeats.

# 20 More periodic features

This chapter describes further periodic features of eProDas and for that matter it cannot be used without comprehensive knowledge of the material in the previous chapter.

## 20.1 Using SPI bus with periodic actions

eProDas supports data exchange over SPI bus during periodic actions. The usual way to realize SPI transfer(s) is to configure SPI module with non-periodic functions (section 15.6.1, for example) and then to add periodic actions for SPI transfer to your periodic programs. Periodic action for reconfiguration of SPI clock also exists as well as shortcuts for using the out-of-the-box supported chips that we have met in the section 17.1.

### 20.1.1 Doing SPI transfer as an atomic operation

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_SPI_Transfer(unsigned int DeviceIdx, unsigned int TxMode,
  unsigned int TxConst, unsigned int CorrectFirstBit, unsigned int EnableRx,
  unsigned int WaitSPI, int TestPoint);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_SPI_Transfer(DeviceIdx: LongWord; TxMode: LongWord;
  TxConst: LongWord; CorrectFirstBit: LongWord; EnableRx: LongWord; WaitSPI: LongWord;
  TestPoint: Integer):integer;
```

Prototypes in VisualBasic

```
eProDas_AddPeriodicAction_SPI_Transfer
```

This function adds periodic action for realizing a SPI transfer of one byte to your periodic program. Parameter `TxMode` determines what you want to send from PIC to SPI periphery. The value of 0 indicates that you do not care about the sent data so eProDas sends out a random value (i.e. eProDas does not waste any time sending a specific value). When the parameter `TxMode` is set to 1 the value of parameter `TxConst` is sent to the periphery (i.e. in each period of execution the same value is transmitted). Finally, when the parameter `TxMode` is set to 2 the value from OUT packet is transmitted and consequently one byte of OUT packet is consumed. This is the usual choice when e.g. implementing signal generation by exciting DA converter with signal pattern of your choice.

The parameter `CorrectFirstBit` determines whether the first bit correction (section 15.6.1) is applied to this transfer (only). Since periodic actions directly synthesize assembly code according to your specifications, the code does not check the state of the first-bit-correction internal variable but instead the proper code for doing the correction is included in the periodic program or skipped altogether, depending of the value of the parameter `CorrectFirstBit`. This way one conditional jump is spared and the code can execute faster. As the section 15.6.1 describes, you should (better: you must, unless you have a strong reason not to) enable the first bit correction (`CorrectFirstBit` = 1) when SPI clock is synthesized with the help of Timer2 and disable it otherwise (`CorrectFirstBit` = 0).

The non-zero value of the parameter `EnableRx` indicates that you want to obtain the value, which SPI periphery sends to the PIC; in this case the value is written to the IN packet (one byte of IN packet is consumed). When the value `EnableRx` is zero, the received value is thrown away.

The parameters `WaitSPI` and `TestPoint` are going to be described further on. For now make sure that the former equals 0 and the latter is set to –1 (yes, set it to a negative number, **not** to zero).

The working of the function is as follows. First, SPI transfer is initiated. Second, eProDas waits in a tight loop that the transfer is over. Third, the transfer is properly finished and the received byte is written to the IN pipe, if so requested.

The duration of SPI transfer is determined by the frequency of SPI clock and it can take a substantial amount of time. For example, with SPI clock of 3 MHz, it takes four ICs to transfer each bit. Consequently, 4 x 8 = 32 ICs pass by before the whole byte is transferred. Besides the raw transfer time eProDas burns certain number of ICs to initiate and finish the transfer, as described further on. According to the description from the previous paragraph, eProDas does not do anything useful during this time but it merely waits for the SPI transfer to finish, which is a shame.

### 20.1.2 Spliting SPI transfer into stages

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_Start_SPI_Transfer(unsigned int DeviceIdx, unsigned int TxMode,
  unsigned int TxConst, unsigned int CorrectFirstBit);

int eProDas_AddPeriodicAction_Wait_SPI_Transfer(unsigned int DeviceIdx);

int eProDas_AddPeriodicAction_Finish_SPI_Transfer(unsigned int DeviceIdx,
  unsigned int EnableRx, int TestPoint);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_Start_SPI_Transfer(DeviceIdx: LongWord; TxMode: LongWord;
  TxConst: LongWord; CorrectFirstBit: LongWord):integer;

function eProDas_AddPeriodicAction_Wait_SPI_Transfer(DeviceIdx: LongWord):integer;

function eProDas_AddPeriodicAction_Finish_SPI_Transfer(DeviceIdx: LongWord;
  EnableRx: LongWord; TestPoint: Integer):integer;
```

Prototypes in VisualBasic

```
eProDas_AddPeriodicAction_Start_SPI_Transfer

eProDas_AddPeriodicAction_Wait_SPI_Transfer

eProDas_AddPeriodicAction_Finish_SPI_Transfer
```

If you are willing to put some effort into optimizing your program, you can squeeze a substantially more performance out of the PIC by splitting the SPI transfer into stages instead of executing it atomically. With function `Start_SPI_Transfer` you only start the SPI transfer, which then progresses in background. Now, instead of merely waiting for its completion, you can do other tasks in parallel, like communicating with other non-SPI chips, reading of comparators, reading of internal AD converter, reading & writing to digital ports, toggling pins and anything else that is not SPI related.

If you manage to occupy the PIC with these additional activities for more time than SPI transfer takes, you can directly `Finish_SPI_Transfer` (this step is **obligatory** even if you do not want to obtain the received byte). However, if other activities take less time or if you are unsure, you can execute `Wait_SPI_Transfer` before finishing it.

The function `Wait_SPI_Transfer` monitors internal SPI state in a tight program loop, so it knows for sure when the transfer is finished. Therefore, the more other activities you execute after you start the SPI transfer but before you `Wait_SPI_Transfer`, the less time of your PIC is going to be wasted on unproductive waiting.

**Note.** Take care to eliminate any possibility that the executing of the activity `Finish_SPI_Transfer` starts before the actual transfer is finished. If you are not sure what you are doing, the action `Wait_SPI_Transfer` is always a safe thing to do, although it consumes a bit of PIC's time for itself.

### 20.1.3 Time consumption of SPI transfer related functions

Okay, we know how to do SPI transfer, now we need to find out how much time these functions take to execute. The execution time of the function `Start_SPI_Transfer` varies according to its parameters. The unavoidable start of SPI transfer in hardware takes 1 IC. Sending of a constant `TxConst` or a byte from OUT packet consumes one additional IC. Correction of the first bit duration consumes 3 additional ICs.

The function `Finish_SPI_Transfer` spends one IC on properly finalizing SPI transfer. If you care about the received byte, one additional IC is spent on writing the value into the IN pipe. If the (not yet described) parameter `TestPoint` is not negative, the function spends two additional ICs.

The function `Wait_SPI_Transfer` takes at least 2 ICs to execute even if SPI transfer is already finished. If SPI transfer is in progress, enough additional delay is inserted that the transfer can finish in peace. After the transfer is finished it takes from 2 to 4 ICs to exit the wait. Although by the last statement it seems that some jitter is introduced into execution by the function `Wait_SPI_Transfer`, this is not necessarily so. Namely, the duration of exit from waiting depends on the state of the program loop at the instant of time when SPI transfer is finished. If things execute equally in each period of execution, then the delay is always the same, since the waiting loop is always at the same stage when transfer finishes.

The described execution times of these SPI related functions must be added to the raw SPI transfer time to get the actual time consumptions. The action `SPI_Transfer`, which does atomic SPI transfer, merely calls the three functions from the section 20.1.2 in a sequence. Consequently, the sum of their execution times plus the duration of actual SPI transfer is the duration of atomic SPI operation.

Of course, it is much more pleasant to rely on periodic analyzer to calculate these figures instead of doing the math by hand, so do not be frightened by the above descriptions.

### 20.1.4 Setting and configuring SPI clock

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_Change_SPI_Clock(unsigned int DeviceIdx, int IN_SampleAtTheEnd,
  int ClockIdleStateHigh, int ClockSpeed, unsigned int ClockPeriod,
  unsigned int ClockPrescaler);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_Change_SPI_Clock(DeviceIdx: LongWord;
  IN_SampleAtTheEnd: Integer; ClockIdleStateHigh: Integer; ClockSpeed: Integer;
  ClockPeriod: LongWord; ClockPrescaler: LongWord):integer;
```

Prototypes in VisualBasic

```
eProDas_AddPeriodicAction_Change_SPI_Clock
```

Section 15.6.6 mentions the need for SPI clock reconfiguration when working with different SPI chips that have their own particular clock needs. Needless to say, we need the possibility to reconfigure SPI clock on the fly during periodic actions. The periodic action `Change_SPI_Clock` works similarly to the function `Configure_SPI_Clock` and the correcponding parameters have a very similar meaning.

The difference is that non-periodic function `Configure_SPI_Clock` always reconfigures all parameters of periodic clock, whereas with periodic action `Change_SPI_Clock` you select which ones of them you want to reconfigure. For that matter note that the parameters `IN_SampleAtTheEnd`, `ClockIdleStateHigh` and `ClockSpeed` are now **signed** integers whereas in the case of the function `Configure_SPI_Clock` they are **unsigned** integers.

When the value of the mentioned parameters is negative, the respective clock setting is not reconfigured to save the execution time. For example, if you are communicating with two SPI chips that both require low idle clock state, you do not have to waste the time on reconfiguration of this parameter, which you achieve by setting the parameter `ClockIdleStateHigh` to a negative value.

When the parameter `ClockSpeed` is negative, the frequency of SPI clock is unchanged and the two other related parameters `ClockPeriod` and `ClockPrescaler` are ignored.

### 20.1.5 Advanced SPI transfer tuning

The basics about periodic SPI transfer are covered now. This section is about squeezing every last possible inch of performance out of SPI transfer, which requires a bit more work from your side than the previous method of split transfers and it provides only a modest increase in performance at best. If you are not willing to bother with miniscule performance gains, which are accompanied with a lot of hard work, skip this section altogether.

Let us start with the action `SPI_Transfer` from section 20.1.1 for doing atomic SPI transfers. The function for adding this action has two parameters `WaitSPI` and `TestPoint`, which have not been described yet.

When the parameter `WaitSPI` is set to zero the function `SPI_Transfer` inserts the function `Wait_SPI_Transfer` after the start of SPI transfer, as it was already described. However, when the parameter `WaitSPI` is set to a non-zero value, the function `Delay` (section 19.12.9) replaces the true waiting phase; the delay in ICs equals the value of the parameter `WaitSPI`. After the delay, the function `Finish_SPI_Transfer` is called in a usual way.

Now, what is the point of all this? If you can specify **the exact** delay so that the function `Finish_SPI_Transfer` executes **precisely** after the actual SPI transfer is finished, you can save 2 to 4 ICs that the function `Wait_SPI_Transfer` would otherwise waste on doing its work.

Okay, I want to do this. So how do I know, what is the proper delay? It is **approximately** equal to the nominal number of ICs that execute during the transfer. Just establish how many ICs it takes to transfer one bit and multiply the figure with 8 to get the result for the whole byte. For example, with SPI clock of 1 MHz it takes 12 ICs to transfer one bit. To transfer the whole byte it takes 12 x 8 = 96 ICs. With SPI clock of 2 MHz, the figure is 48 ICs, accordingly. When you have the number of burned ICs at hand, set the parameter `WaitSPI` to that number and you are **close to** the proper value. You have just completed the first phase of the tedious tuning process.

Now, eProDas will help you to get **the exact** needed delay. For that matter set the parameter `TestPoint` to some value between 0 and 7. When you do this, the function `Finish_SPI_Transfer` (which is called automatically by the function `SPI_Transfer`) inserts a small piece of code (which burns two ICs) into your program to check whether SPI transfer really finished already and if not, it sets the selected error indicator (from 0 to 7) to a non-zero value. These indicators are simply bits of a one-byte variable that eProDas device automatically resets upon the start of periodic actions. If all SPI transfers that are monitored with test points execute well, then the control variable remains at zero; otherwise its appropriate bits are set.

For example, by violating SPI transfer monitored by test point 0, the bit zero of the error variable would be set and the value would be 1. Similarly, test point 4 (or 7) sets the fourth (or seventh) bit so the value of variable becomes 16 (or 128). By selecting different test points for monitoring different SPI transfers you can debug up to eight transfers in parallel. Note that it is not an error to monitor different transfers with the same test point, but this way you can only know that at least one of them was violated but you cannot deduce which one(s) was(were) responsible for setting the indicator.

So, the algorithm for tuning the SPI transfer is as follows. Set `WaitSPI` to the calculated delay and set `TestPoint` to the test point of your selection. Run your periodic program for as long as you want. **After** you terminate execution of periodic program by calling the function `StopPeriodicActions`, you can read the value of all test points with the following function.

#### 20.1.5.1 ReadTestPoints

Prototype in C/C++

```
int eProDas_ReadTestPoints(unsigned int DeviceIdx, unsigned int &TestPoints);
```

Prototype in Delphi

```
function eProDas_ReadTestPoints(DeviceIdx: LongWord;
  var TestPoints: LongWord):integer;
```

Prototype in VisualBasic

```
eProDas_ReadTestPoints
```

The value of all test points is returned through the parameter `TestPoints`. If this parameter is zero, all tested delays are long enough and there are no violations of SPI laws. The question is, whether we could shorten the delay to gain more speed. For that matter try to slightly decrease delay(s), specified by `WaitSPI` and run the program again. As long as test points remain at zero, the delay is sufficiently long. The most appropriate delay is the smallest one that does not set any test points. When you have finished deduced these numbers remove test points from your code (by setting all `TestPoint` parameters to a negative value) in order not to waste any ICs on the unnecessary checking, and feel confident that you have the fastest possible SPI transfer on the PIC18F4550's planet.

The same optimization technique can be applied to the split SPI transfer, except you should shorten the `WaitSPI` delays further for the amount of time that other activities are spending in parallel with SPI transfer. First, start SPI transfer, and then do whatever you want to do in parallel with it. Now, instead of waiting for SPI transfer to finish (with action `Wait_SPI_Transfer`), insert a `Delay` (section 19.12.9) of an appropriate length. Initially, just make a guest and see what periodic analyzer has to say about your program. Then adjust delays according to its suggestions and start a real-time debugging with test points technique. Never trust the analyzer since it is even conceptually too simple to always guess the proper figures. Test points are true data from the field and whatever they reveal, it is the truth.

## 20.2 Out-of-the-box support for SPI attached periphery

Similarly to the non-periodic out-of-the-box support for certain SPI chips (section 17.1) eProDas eases your working with the same set of periphery when constellating periodic programs. On a conceptual level these API functions are similar to the generic SPI transfer by means of giving you a choice whether you want an easy development or hardworking tuning of SPI communication.

### 20.2.1 Communication with AD converter MCP320x from Microchip

According to the previously described philosophy, the actions for periodic communication with MCP320x enable you to do either an easy atomic SPI communication or a bit harder-to-configure but potentially more efficient split transfer.

#### 20.2.1.1 Atomic communication with AD converter MCP320x

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_MCP3208(unsigned int DeviceIdx, unsigned int Channel,
  unsigned int SingleEnded, unsigned int ChipSelectPort, unsigned int ChipSelectPin,
  unsigned int CorrectFirstBit, unsigned int WaitSPI, int TestPoint);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_MCP3208(DeviceIdx: LongWord; Channel: LongWord;
  SingleEnded: LongWord; ChipSelectPort: LongWord; ChipSelectPin: LongWord;
  CorrectFirstBit: LongWord; WaitSPI: LongWord; TestPoint: Integer):integer;
```

Prototypes in VisualBasic

```
eProDas_AddPeriodicAction_MCP3208
```

**Note.** Do not be confused by the action name …MCP3208. You can use this and the following actions for both MCP3208 as well as MCP3204 chips.

Parameters should be already familiar, since they are either equal to the ones that non-periodic function ReadExternalAD_MCP320x possesses (section 17.1.1.2) or they were described previously in this chapter. There are two exceptions from the already described functionality, though.

The parameter WaitSPI has a slightly expanded meaning: if you set this parameter to the value of 1, the appropriate delay is calculated for you automatically. This works only if you are using the prescribed SPI clock of 2 MHz, otherwise you can do a safe waiting (WaitSPI value of 0) or prescribe the delay by yourself (WaitSPI value greater than 1 still means the delay in ICs).

If Channel is less than or equal to 7, then this parameter specifies the channel to be read in a usual way. However, when the Channel is greater than 7, the actual channel number and regime are read from the OUT pipe (the parameter SingleEnded is ignored in this case), which enables the flexibility of reading channels in an arbitrary way. Note that you cannot simply put the plain channel number into the OUT packet, but you must instead insert the value that the following function synthesizes.

#### 20.2.1.2 Calc_MCP3208_ChannelCode

Prototype in C/C++

```
int eProDas_Calc_MCP3208_ChannelCode(unsigned int Channel,
  unsigned int SingleEnded, unsigned char &Code);
```

Prototype in Delphi

```
function eProDas_Calc_MCP3208_ChannelCode(Channel: LongWord; SingleEnded: LongWord;
  var Code: Byte):integer;
```

Prototype in VisualBasic

```
eProDas_Calc_MCP3208_ChannelCode
```

Regardless of the channel selection method, the result of AD conversion is written to the IN pipe as two bytes. The lower nibble of the first byte represents the four MSB bits of the result whereas the next byte contains the eight LSB bits of the result. The upper nibble of the first byte contains random data so you must no assume that it is zero when building AD result.

Instead of doing bit shifting and logic operations by yourself, you can use the following eProDas functions for calculating the whole 12-bit AD result.

### 20.2.1.3 MCP3208_BuildResult (_Advance)

Prototype in C/C++

```
unsigned int eProDas_MCP3208_BuildResult(unsigned char *Data);

unsigned int eProDas_MCP3208_BuildResult_Advance(unsigned char *&Data);
```

Prototype in Delphi

```
function eProDas_MCP3208_BuildResult(Data: PByte):longword;

function eProDas_MCP3208_BuildResult_Advance(var Data: PByte):longword;
```

Prototype in VisualBasic

```
eProDas_MCP3208_BuildResult

eProDas_MCP3208_BuildResult_Advance
```

These two functions never return any error codes so they can deliver the result through the return value instead of through some pass-by-reference parameter. The `Data` is a pointer to the first byte of the IN packet that contains the AD result. In the case of the function `MCP3208_BuildResult_Advance` the `Data` pointer is automatically advanced for two bytes which is handy if you use this pointer to scan the whole IN packet buffer.

### 20.2.1.4 Configuring SPI clock for AD converter MCP320x

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_SPI_Clock_MCP3208(unsigned int DeviceIdx,
  unsigned int ReconfigFrequency, unsigned int ReconfigSampleInstant,
  unsigned int ReconfigIdleState);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_SPI_Clock_MCP3208(DeviceIdx: LongWord;
  ReconfigFrequency: LongWord; ReconfigSampleInstant: LongWord;
  ReconfigIdleState: LongWord):integer;
```

Prototypes in VisualBasic

```
eProDas_AddPeriodicAction_SPI_Clock_MCP3208
```

Whenever you need to reconfigure SPI clock to work with MCP3208, you can use this action. The three parameters do not specify configuration values which are instead deduced automatically, but their non-zero values only instruct that certain parameter of SPI clock is reconfigured: `ReconfigFrequency` enables or inhibits reconfiguration of frequency (2 MHz), whereas the latter two parameters determine whether the configuration of sampling instant and clock idle state, respectively, should take place.

If hard SPI tuning is your hobby, you should always select the minimal reconfiguration path, which saves execution time. On the other hand, if you want to be on a safe side, always reconfigure everything. Generally, the sampling instant does not need reconfiguration, since all SPI chips that eProDas supports use the same setting of this parameter.

### 20.2.1.5 About the performance of MCP320x

Note that the maximal SPI clock that MCP320x supports is 2 MHz so it takes 48 ICs to transfer one byte worth of data. Further, 3 bytes must be transferred between PIC and SPI for obtaining one AD result since besides the 12-bit result the channel and single ended or differential mode of operation must be specified. It takes slightly more than 150 ICs to progress the entire procedure, so this is a nice opportunity to do a lot of other work in parallel by splitting the MCP320x operation in several parts.

### 20.2.1.5.1 Split communication with AD converter MCP320x

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_MCP3208_Part1(unsigned int DeviceIdx, unsigned int Channel,
  unsigned int SingleEnded, unsigned int ChipSelectPort, unsigned int ChipSelectPin,
  unsigned int CorrectFirstBit);

int eProDas_AddPeriodicAction_MCP3208_Part2(unsigned int DeviceIdx, unsigned int Channel,
  unsigned int CorrectFirstBit, int TestPoint);

int eProDas_AddPeriodicAction_MCP3208_Part3(unsigned int DeviceIdx,
  unsigned int CorrectFirstBit, int TestPoint);

int eProDas_AddPeriodicAction_MCP3208_Part4(unsigned int DeviceIdx,
  unsigned int ChipSelectPort, unsigned int ChipSelectPin, int TestPoint);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_MCP3208_Part1(DeviceIdx: LongWord; Channel: LongWord;
  SingleEnded: LongWord; ChipSelectPort: LongWord; ChipSelectPin: LongWord;
  CorrectFirstBit: LongWord):integer;

function eProDas_AddPeriodicAction_MCP3208_Part2(DeviceIdx: LongWord; Channel: LongWord;
  CorrectFirstBit: LongWord; TestPoint: Integer):integer;

function eProDas_AddPeriodicAction_MCP3208_Part3(DeviceIdx: LongWord;
  CorrectFirstBit: LongWord; TestPoint: Integer):integer;

function eProDas_AddPeriodicAction_MCP3208_Part4(DeviceIdx: LongWord;
  ChipSelectPort: LongWord; ChipSelectPin: LongWord; TestPoint: Integer):integer;
```

Prototypes in VisualBasic

```
eProDas_AddPeriodicAction_MCP3208_Part1

eProDas_AddPeriodicAction_MCP3208_Part2

eProDas_AddPeriodicAction_MCP3208_Part3

eProDas_AddPeriodicAction_MCP3208_Part4
```

When splitting MCP3208 transfer the same logic applies as in the case of a split generic SPI transfer. Part 1 initiates SPI transfer of the first byte; in this stage one byte of OUT packet is consumed if the target channel is read from the OUT pipe.

Now you have about 48 ICs to do some other work. Then you can either wait on completion of the transfer or make a precise delay with a help of testpoints; you must manually insert delay or waiting action before part 2 if needed. Then part 2 finishes the transfer of the first byte and initiates the transfer of the second one. In this stage the received byte over SPI is discarded, since it does not carry any part of the AD result. Again, after part 2 there about 48 ICs for other tasks (or wait/delay).

Part 3 finishes transfer of the second byte and starts transfer of the third one. The second byte is written to the IN pipe; its low nibble contains the MSB bits of the AD result. Then after a useful work of 48 ICs, a wait or a delay, part 4 finishes the transfer of the last byte and writes the result into the IN pipe. This byte holds the lowest 8 bits of AD result.

**Note.** If some periodic actions that consume IN pipe are inserted between execution of parts 3 and 4, the two bytes that hold AD result are not placed adjacently into the IN pipe any more. Consequently, you cannot use the functions `MCP3208_BuildResult` (`_Advance`) to build the result, but you must do it either by hand or you can use the following function by providing values of the two result bytes manually.

#### 20.2.1.5.2 MCP3208_BuildResult_Separated

Prototypes in C/C++

```
unsigned int eProDas_MCP3208_BuildResult_Separated(unsigned char FirstByte,
  unsigned char SecondByte);
```

Prototypes in Delphi

```
function eProDas_MCP3208_BuildResult_Separated(FirstByte: Byte;
  SecondByte: Byte):longword;
```

Prototypes in VisualBasic

```
eProDas_MCP3208_BuildResult_Separated
```

The `FirstByte` and `SecondByte` must be set to the values that the parts 3 and 4, respectively, write to the IN pipe during periodic execution.

### 20.2.1.6 Atomic communication with MCP320x for obtaining 9-bit result

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_MCP3208_9bit(unsigned int DeviceIdx, unsigned int Channel,
  unsigned int SingleEnded, unsigned int ChipSelectPort, unsigned int ChipSelectPin,
  unsigned int CorrectFirstBit, unsigned int WaitSPI, int TestPoint);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_MCP3208_9bit(DeviceIdx: LongWord; Channel: LongWord;
  SingleEnded: LongWord; ChipSelectPort: LongWord; ChipSelectPin: LongWord;
  CorrectFirstBit: LongWord; WaitSPI: LongWord; TestPoint: Integer):integer;
```

Prototypes in VisualBasic

```
eProDas_AddPeriodicAction_MCP3208_9bit
```

Sometimes the speed of execution is more important than the precision of AD result. In the case of the chip MCP320x, we can save one SPI transfer out of three ones if 9-bit result is enough for our scenario of usage. The option is attractive since it can be combined with full precision reading in an arbitrary way. Therefore, if you need to read, say, 2 inputs with full precision but the rest are not so critical, you can intermix the precision of AD results reading accordingly.

To build a proper value to be written to the OUT pipe for selecting AD channel and regime of operation (when the parameter `Channel` is greater than 7), use the following function.

### 20.2.1.7 Calc_MCP3208_9bit_ChannelCode

Prototype in C/C++

```
int eProDas_Calc_MCP3208_9bit_ChannelCode(unsigned int Channel,
  unsigned int SingleEnded, unsigned char &Code);
```

Prototype in Delphi

```
function eProDas_Calc_MCP3208_9bit_ChannelCode(Channel: LongWord;
  SingleEnded: LongWord; var Code: Byte):integer;
```

Prototype in VisualBasic

```
eProDas_Calc_MCP3208_9bit_ChannelCode
```

Regardless of the channel selection method, the result is still delivered as two separate bytes of which the first one contains the MSB bit of the result whereas the second one contains the rest of the value. You can use the following two functions for constellating the result (between 0 and 511, not between 0 and 4095) from the two pieces.

### 20.2.1.8 MCP3208_9bit_BuildResult (_Advance)

Prototype in C/C++

```
unsigned int eProDas_MCP3208_9bit_BuildResult(unsigned char *Data);

unsigned int eProDas_MCP3208_9bit_BuildResult_Advance(unsigned char *&Data);
```

Prototype in Delphi

```
function eProDas_MCP3208_9bit_BuildResult(Data: PByte):longword;

function eProDas_MCP3208_9bit_BuildResult_Advance(var Data: PByte):longword;
```

Prototype in VisualBasic

```
eProDas_MCP3208_9bit_BuildResult

eProDas_MCP3208_9bit_BuildResult_Advance
```

Although 9-bit operation of MCP320x saves one third of the time, you may still want to realize it as a separate SPI transfer in order to execute other tasks in parallel. Here are the functions for achieving this functionality.

### 20.2.1.9 Split communication with MCP320x for obtaining 9-bit result

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_MCP3208_8bit_Part1(unsigned int DeviceIdx, unsigned int Channel,
  unsigned int SingleEnded, unsigned int ChipSelectPort, unsigned int ChipSelectPin,
  unsigned int CorrectFirstBit);

int eProDas_AddPeriodicAction_MCP3208_8bit_Part2(unsigned int DeviceIdx,
  unsigned int CorrectFirstBit, int TestPoint);

int eProDas_AddPeriodicAction_MCP3208_8bit_Part3(unsigned int DeviceIdx,
  unsigned int ChipSelectPort, unsigned int ChipSelectPin, int TestPoint);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_MCP3208_9bit_Part1(DeviceIdx: LongWord; Channel: LongWord;
  SingleEnded: LongWord; ChipSelectPort: LongWord; ChipSelectPin: LongWord;
  CorrectFirstBit: LongWord):integer;

function eProDas_AddPeriodicAction_MCP3208_9bit_Part2(DeviceIdx: LongWord;
  CorrectFirstBit: LongWord; TestPoint: Integer):integer;

function eProDas_AddPeriodicAction_MCP3208_9bit_Part3(DeviceIdx: LongWord;
  ChipSelectPort: LongWord; ChipSelectPin: LongWord; TestPoint: Integer):integer;
```

Prototypes in VisualBasic

```
eProDas_AddPeriodicAction_MCP3208_9bit_Part1

eProDas_AddPeriodicAction_MCP3208_9bit_Part2

eProDas_AddPeriodicAction_MCP3208_9bit_Part3
```

Split transfer for obtaining 9-bit AD result is done in a similar way to the one for 12-bit result (section 20.2.1.5.1), except that this time we only need to execute 3 parts instead of 4 ones. Part 1 initiates the first SPI transfer (and consumes one byte of OUT pipe if the target channel is not constant). Part 2 finishes the first transfer and initialtes the second one; also the received byte is written to the IN pipe. Part 3 finishes the second transfer and again writes the received byte to the IN pipe.

**Note.** If some periodic actions that consume IN pipe are inserted between execution of parts 2 and 3, the two bytes that hold AD result are not placed adjacently into the IN pipe any more. Consequently, you cannot use the functions `MCP3208_9bit_BuildResult` (`_Advance`) to build the result, but you must do it either by hand or you can use the following function by providing values of the two result bytes manually.

#### 20.2.1.9.1 MCP3208_9bit_BuildResult_Separated

Prototypes in C/C++
```
unsigned int eProDas_MCP3208_9bit_BuildResult_Separated(unsigned char FirstByte,
  unsigned char SecondByte);
```

Prototypes in Delphi
```
function eProDas_MCP3208_9bit_BuildResult_Separated(FirstByte: Byte;
  SecondByte: Byte):longword;
```

Prototypes in VisualBasic
```
eProDas_MCP3208_9bit_BuildResult_Separated
```

The `FirstByte` and `SecondByte` must be set to the values that the parts 2 and 3, respectively, write to the IN pipe during periodic execution.

### 20.2.1.10 Atomic communication with MCP320x for obtaining 8-bit result

Prototypes in C/C++
```
int eProDas_AddPeriodicAction_MCP3208_8bit(unsigned int DeviceIdx, unsigned int Channel,
  unsigned int SingleEnded, unsigned int ChipSelectPort, unsigned int ChipSelectPin,
  unsigned int CorrectFirstBit, unsigned int WaitSPI, int TestPoint);
```

Prototypes in Delphi
```
function eProDas_AddPeriodicAction_MCP3208_8bit(DeviceIdx: LongWord; Channel: LongWord;
  SingleEnded: LongWord; ChipSelectPort: LongWord; ChipSelectPin: LongWord;
  CorrectFirstBit: LongWord; WaitSPI: LongWord; TestPoint: Integer):integer;
```

Prototypes in VisualBasic
```
eProDas_AddPeriodicAction_MCP3208_8bit
```

Besides the speed of execution you can save one byte of IN packet if 8-bit AD result offers enough precision for your application. Again, the option is attractive since it can be combined with full precision or 9-bit reading in an arbitrary way.

To build a proper value to be written to the OUT pipe for selecting AD channel and regime of operation (when the parameter `Channel` is greater than 7), use the following function.

### 20.2.1.11 Calc_MCP3208_8bit_ChannelCode

Prototype in C/C++
```
int eProDas_Calc_MCP3208_8bit_ChannelCode(unsigned int Channel,
  unsigned int SingleEnded, unsigned char &Code);
```

Prototype in Delphi
```
function eProDas_Calc_MCP3208_8bit_ChannelCode(Channel: LongWord;
  SingleEnded: LongWord; var Code: Byte):integer;
```

Prototype in VisualBasic
```
eProDas_Calc_MCP3208_8bit_ChannelCode
```

Regardless of the channel selection method, the result is delivered in one IN byte, so in this case the functions for building the result from the two pieces do not make much sense. However, for the sake of orthogonality, the following two functions are provided by eProDas API.

### 20.2.1.12 MCP3208_8bit_BuildResult (_Advance)

Prototype in C/C++

```
unsigned int eProDas_MCP3208_8bit_BuildResult(unsigned char *Data);

unsigned int eProDas_MCP3208_8bit_BuildResult_Advance(unsigned char *&Data);
```

Prototype in Delphi

```
function eProDas_MCP3208_8bit_BuildResult(Data: PByte):longword;

function eProDas_MCP3208_8bit_BuildResult_Advance(var Data: PByte):longword;
```

Prototype in VisualBasic

```
eProDas_MCP3208_8bit_BuildResult

eProDas_MCP3208_8bit_BuildResult_Advance
```

The first function merely returns the value that the pointer Data points to, whereas the latter also advances the pointer for one byte. There are no other effects so you can probably produce a more efficient code if you skip these two functions altogether.

### 20.2.1.13 Split communication with MCP320x for obtaining 8-bit result

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_MCP3208_8bit_Part1(unsigned int DeviceIdx, unsigned int Channel,
  unsigned int SingleEnded, unsigned int ChipSelectPort, unsigned int ChipSelectPin,
  unsigned int CorrectFirstBit);

int eProDas_AddPeriodicAction_MCP3208_8bit_Part2(unsigned int DeviceIdx,
  unsigned int CorrectFirstBit, int TestPoint);

int eProDas_AddPeriodicAction_MCP3208_8bit_Part3(unsigned int DeviceIdx,
  unsigned int ChipSelectPort, unsigned int ChipSelectPin, int TestPoint);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_MCP3208_8bit_Part1(DeviceIdx: LongWord; Channel: LongWord;
  SingleEnded: LongWord; ChipSelectPort: LongWord; ChipSelectPin: LongWord;
  CorrectFirstBit: LongWord):integer;

function eProDas_AddPeriodicAction_MCP3208_8bit_Part2(DeviceIdx: LongWord;
  CorrectFirstBit: LongWord; TestPoint: Integer):integer;

function eProDas_AddPeriodicAction_MCP3208_8bit_Part3(DeviceIdx: LongWord;
  ChipSelectPort: LongWord; ChipSelectPin: LongWord; TestPoint: Integer):integer;
```

Prototypes in VisualBasic

```
eProDas_AddPeriodicAction_MCP3208_8bit_Part1

eProDas_AddPeriodicAction_MCP3208_8bit_Part2

eProDas_AddPeriodicAction_MCP3208_8bit_Part3
```

Split transfer for obtaining 8-bit AD result is done in a completely the same way as the one for obraining 9-bit result (section 20.2.1.9). Part 1 initiates the first SPI transfer (and consumes one OUT byte if the channel is not prescribed in advance). Part 2 finishes the first transfer and initialtes the second one (in comparison to the 9-bit AD reading no byte is written ot the IN pipe at this stage). Part 3 finishes the second transfer and writes the whole 8-bit result to the IN pipe.

## 20.2.2 Communication with AD converter ADS7816 from Texas Instruments

The actions for periodic communication with ADS7816 again enable you to do either an easy atomic SPI communication or a bit harder-to-configure but potentially more efficient split transfer.

#### 20.2.2.1 Atomic communication with AD converter ADS7816

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_ADS7816(unsigned int DeviceIdx, unsigned int ChipSelectPort,
  unsigned int ChipSelectPin, unsigned int CorrectFirstBit, unsigned int WaitSPI,
  int TestPoint);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_ADS7816(DeviceIdx: LongWord; ChipSelectPort: LongWord;
  ChipSelectPin: LongWord; CorrectFirstBit: LongWord; WaitSPI: LongWord;
  TestPoint: Integer):integer;
```

Prototypes in VisualBasic

```
eProDas_AddPeriodicAction_ADS7816
```

Parameters should be already familiar, since they are either equal to the ones that non-periodic function `ReadExternalAD_ADS7816` possesses (section 17.1.2.2) or they were described previously in this chapter. Similarly to the working of actions for MCP320x the meaning of the parameter `WaitSPI` is expanded: when its value is 1, the appropriate delay is calculated for you automatically. This works only if you are using the prescribed SPI clock of 3 MHz, otherwise you can do a safe waiting (`WaitSPI` value of 0) or prescribe the delay by yourself (`WaitSPI` value greater than 1 still means the delay in ICs).

The result is delivered as two separate bytes of which the first one contains five MSB bits of the result whereas bits 7:1 of the second byte contain the rest of the result. Other bits are unspecified and you must not assume that they are zero. You can use the following two functions for constellating the result from the two pieces.

#### 20.2.2.2 ADS7816_BuildResult (_Advance)

Prototype in C/C++

```
unsigned int eProDas_ADS7816_BuildResult(unsigned char *Data);

unsigned int eProDas_ADS7816_BuildResult_Advance(unsigned char *&Data);
```

Prototype in Delphi

```
function eProDas_ADS7816_BuildResult(Data: PByte):longword;

function eProDas_ADS7816_BuildResult_Advance(var Data: PByte):longword;
```

Prototype in VisualBasic

```
eProDas_ADS7816_BuildResult

eProDas_ADS7816_BuildResult_Advance
```

These two functions never return any error codes so they can deliver the result through the return value instead of through a pass-by-reference parameter. The `Data` is a pointer to the first byte of the IN packet that contains the AD result. The `Advance` version of the function also advances the `Data` pointer for two bytes.

### 20.2.2.3 Configuring SPI clock for ADS7816

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_SPI_Clock_ADS7816(unsigned int DeviceIdx,
  unsigned int ReconfigFrequency, unsigned int ReconfigSampleInstant,
  unsigned int ReconfigIdleState);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_SPI_Clock_ADS7816(DeviceIdx: LongWord;
  ReconfigFrequency: LongWord; ReconfigSampleInstant: LongWord;
  ReconfigIdleState: LongWord):integer;
```

Prototypes in VisualBasic

```
eProDas_AddPeriodicAction_SPI_Clock_ADS7816
```

Whenever you need to reconfigure SPI clock to work with ADS7816, you can use this action. The three parameters do not specify configuration values which are instead deduced automatically, but their non-zero values only instruct that certain parameter of SPI clock is reconfigured: `ReconfigFrequency` enables or inhibits reconfiguration of frequency (3 MHz), whereas the latter two parameters determine whether the configuration of sampling instant and clock idle state, respectively, should take place.

If hard SPI tuning is your hobby, you should always select the minimal reconfiguration path, which saves execution time. On the other hand, if you want to be on a safe side, always reconfigure everything. Generally, the sampling instant does not need reconfiguration, since all SPI chips that eProDas supports use the same setting of this parameter.

### 20.2.2.4 About the performance of ADS7816

The maximal SPI clock that ADS7816 supports is 3.2 MHz, but PIC can only synthesize SPI clock of 3 MHz. So, it takes 32 ICs to transfer one byte worth of data. For each AD reading 2 bytes must be transferred to be able to deliver 12-bit result. It takes about 76 ICs to progress the entire procedure, so this is a nice opportunity to do a lot of other work in parallel by splitting the ADS7816 operation in several parts.

#### 20.2.2.4.1 Split communication with AD converter ADS7816

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_ADS7816_Part1(unsigned int DeviceIdx,
  unsigned int ChipSelectPort, unsigned int ChipSelectPin, unsigned int CorrectFirstBit);

int eProDas_AddPeriodicAction_ADS7816_Part2(unsigned int DeviceIdx,
  unsigned int CorrectFirstBit, int TestPoint);

int eProDas_AddPeriodicAction_ADS7816_Part3(unsigned int DeviceIdx,
  unsigned int ChipSelectPort, unsigned int ChipSelectPin, int TestPoint);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_ADS7816_Part1(DeviceIdx: LongWord;
  ChipSelectPort: LongWord; ChipSelectPin: LongWord; CorrectFirstBit: LongWord):integer;

function eProDas_AddPeriodicAction_ADS7816_Part2(DeviceIdx: LongWord;
  CorrectFirstBit: LongWord; TestPoint: Integer):integer;

function eProDas_AddPeriodicAction_ADS7816_Part3(DeviceIdx: LongWord;
  ChipSelectPort: LongWord; ChipSelectPin: LongWord; TestPoint: Integer):integer;
```

Prototypes in VisualBasic

```
eProDas_AddPeriodicAction_ADS7816_Part1

eProDas_AddPeriodicAction_ADS7816_Part2

eProDas_AddPeriodicAction_ADS7816_Part4
```

When splitting ADS7816 transfer the same logic applies as in the case of a split generic SPI transfer. Part 1 initiates SPI transfer of the first byte. Now you have about 32 ICs to do some other work. Then you can either wait on completion of the transfer or make a precise delay with a help of testpoints; you must manually insert delay or waiting action before part 2, if needed. Then part 2 finishes the transfer of the first byte and initiates the transfer of the second one; the received byte from the first SPI transfer is written to the IN pipe.

Again, after part 2 there about 32 ICs for other tasks (or wait/delay). Part 3 finishes transfer of the second byte. The received byte is again written to the IN pipe.

**Note.** If some periodic actions that consume IN pipe are inserted between execution of parts 2 and 3, the two bytes that hold AD result are not placed adjacently into the IN pipe any more. Consequently, you cannot use the functions `ADS7816_BuildResult` (`_Advance`) to build the result, but you must do it either by hand or you can use the following function by providing values of the two result bytes manually.

#### 20.2.2.4.2 ADS7816_BuildResult_Separated

Prototypes in C/C++

```
unsigned int eProDas_ADS7816_BuildResult_Separated(unsigned char FirstByte,
  unsigned char SecondByte);
```

Prototypes in Delphi

```
function eProDas_ADS7816_BuildResult_Separated(FirstByte: Byte;
  SecondByte: Byte):longword;
```

Prototypes in VisualBasic

```
eProDas_ADS7816_BuildResult_Separated
```

The `FirstByte` and `SecondByte` must be set to the values that the parts 2 and 3, respectively, write to the IN pipe during periodic execution.

### 20.2.3 Communication with DA converter DAC7611 from Texas Instruments

The actions for periodic communication with DAC7611 again enable you to do either an easy atomic SPI communication or a bit harder-to-configure but potentially more efficient split transfer.

#### 20.2.3.1 Atomic communication with DA converter DAC7611

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_DAC7611(unsigned int DeviceIdx, unsigned int Value,
  unsigned int ChipSelectPort, unsigned int ChipSelectPin, unsigned int LoadPort,
  unsigned int LoadPin, unsigned int CorrectFirstBit, unsigned int WaitSPI, int TestPoint);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_DAC7611(DeviceIdx: LongWord; Value: LongWord;
  ChipSelectPort: LongWord; ChipSelectPin: LongWord; LoadPort: LongWord; LoadPin: LongWord;
  CorrectFirstBit: LongWord; WaitSPI: LongWord; TestPoint: Integer):integer;
```

Prototypes in VisualBasic

```
eProDas_AddPeriodicAction_DAC7611
```

Parameters should be already familiar, since they are either equal to the ones that non-periodic function `WriteExternalDA_DAC7611` possesses (section 17.1.3.2) or they were described previously in this chapter. The meaning of the parameter `WaitSPI` is again broadened in comparison to the generic SPI transfer: when its value is 1, the appropriate delay is calculated for you automatically. This works only if you are using the prescribed SPI clock of 12 MHz, otherwise you can do a safe waiting (`WaitSPI` value of 0) or prescribe the delay by yourself (`WaitSPI` value greater than 1 still means the delay in ICs).

When the parameter `Value` is between 0 and 4095 its value is written to the DAC (i.e. during each period of execution the same value is written to the chip). However, when the `Value` is greater than 4095, then the value from OUT pipe (two bytes) is used; use the following two functions for calculating proper OUT entries.

#### 20.2.3.2 DAC7611_BuildValue (_Advance)

Prototype in C/C++

```
int eProDas_DAC7611_BuildValue(unsigned int Value, unsigned char *Data);

int eProDas_DAC7611_BuildValue_Advance(unsigned int Value, unsigned char *&Data);
```

Prototype in Delphi

```
function eProDas_DAC7611_BuildValue(Value: LongWord; Data: PByte):integer;

function eProDas_DAC7611_BuildValue_Advance(Value: LongWord;
  var Data: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_DAC7611_BuildValue

eProDas_DAC7611_BuildValue_Advance
```

The value to be written to DAC is specified by the `Value` parameter; this time an error occurs if the value is greater than 4095. The two resulting bytes are written to the buffer starting at location `Data`. The `Advance` version of the function also advances the `Data` pointer for two bytes afterwards.

#### 20.2.3.3 Configuring SPI clock for DAC7611

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_SPI_Clock_DAC7611(unsigned int DeviceIdx,
  unsigned int ReconfigFrequency, unsigned int ReconfigSampleInstant,
  unsigned int ReconfigIdleState);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_SPI_Clock_DAC7611(DeviceIdx: LongWord;
  ReconfigFrequency: LongWord; ReconfigSampleInstant: LongWord;
  ReconfigIdleState: LongWord):integer;
```

Prototypes in VisualBasic

```
eProDas_AddPeriodicAction_SPI_Clock_DAC7611
```

Whenever you need to reconfigure SPI clock to work with DAC7611, you can use this action. The three parameters do not specify configuration values which are instead deduced automatically, but their non-zero values only instruct that certain parameter of SPI clock is reconfigured: `ReconfigFrequency` enables or inhibits reconfiguration of frequency (12 MHz), whereas the latter two parameters determine whether the configuration of sampling instant and clock idle state, respectively, should take place.

If hard SPI tuning is your hobby, you should always select the minimal reconfiguration path, which saves execution time. On the other hand, if you want to be on a safe side, always reconfigure everything. Generally, the sampling instant does not need reconfiguration, since all SPI chips that eProDas supports use the same setting of this parameter.

### 20.2.3.4 About the performance of DAC7611

The maximal SPI clock that DAC7611 supports is 20 MHz, but PIC can only synthesize SPI clock of 12 MHz. So, it takes 8 ICs (actually, the test point examination reveals 9 ICs) to transfer one byte worth of data. For each DA writing 2 bytes must be transferred to be able to work with 12-bit precision. Since SPI clock is set to maximal frequency of 12 MHz there is substantially less time to do other things in parallel with SPI transfer than there was when working with some other out-of-the-box chips. Nonetheless, eProDas enables you to do a split DAC7611 transfer to gain even such miniscule opportunities for eProDas performance increase.

#### 20.2.3.4.1 Split communication with DA converter DAC7611

Prototypes in C/C++

```
int eProDas_AddPeriodicAction_DAC7611_Part1(unsigned int DeviceIdx, unsigned int Value,
  unsigned int ChipSelectPort, unsigned int ChipSelectPin, unsigned int CorrectFirstBit);

int eProDas_AddPeriodicAction_DAC7611_Part2(unsigned int DeviceIdx, unsigned int Value,
  unsigned int CorrectFirstBit, int TestPoint);

int eProDas_AddPeriodicAction_DAC7611_Part3(unsigned int DeviceIdx,
  unsigned int ChipSelectPort, unsigned int ChipSelectPin, unsigned int LoadPort,
  unsigned int LoadPin, int TestPoint);
```

Prototypes in Delphi

```
function eProDas_AddPeriodicAction_DAC7611_Part1(DeviceIdx: LongWord; Value: LongWord;
  ChipSelectPort: LongWord; ChipSelectPin: LongWord; CorrectFirstBit: LongWord):integer;

function eProDas_AddPeriodicAction_DAC7611_Part2(DeviceIdx: LongWord; Value: LongWord;
  CorrectFirstBit: LongWord; TestPoint: Integer):integer;

function eProDas_AddPeriodicAction_DAC7611_Part3(DeviceIdx: LongWord;
  ChipSelectPort: LongWord; ChipSelectPin: LongWord; LoadPort: LongWord; LoadPin: LongWord;
  TestPoint: Integer):integer;
```

Prototypes in VisualBasic

```
eProDas_AddPeriodicAction_DAC7611_Part1

eProDas_AddPeriodicAction_DAC7611_Part2

eProDas_AddPeriodicAction_DAC7611_Part4
```

When splitting DAC7611 transfer the same logic applies as in the case of a split generic SPI transfer. Part 1 initiates SPI transfer of the first byte. If the parameter `Value` is greater than 4095, then one byte from OUT pipe is consumed in this phase. Now you have 9 ICs to do some other work. Then you can either wait on completion of the transfer or make a precise delay with a help of testpoints; you must manually insert delay or waiting action before part 2, if needed. Then part 2 finishes the transfer of the first byte and initiates the transfer of the second one. Again, if the parameter `Value` is greater than 4095, then another one byte from OUT pipe is consumed in this phase. After part 2 there about 9 ICs for other tasks (or wait/delay). Part 3 merely finishes transfer of the second byte.

**Note.** If some periodic actions that consume OUT pipe are inserted between execution of parts 1 and 2, the two bytes that hold DA value are not placed adjacently into the OUT pipe any more. Consequently, you cannot use the functions `DAC7611_BuildValue` (`_Advance`) to writte the proper OUT contents into the bufer (directly). The following function which returns the caluclated OUT contents as a separate bytes may be of an assistance in such cases.

### 20.2.3.4.2 DAC7611_BuildValue_Separated

Prototypes in C/C++

```
int eProDas_DAC7611_BuildValue_Separated(unsigned int Value, unsigned char &Byte1,
  unsigned char &Byte2);
```

Prototypes in Delphi

```
function eProDas_DAC7611_BuildValue_Separated(Value: LongWord; var Byte1: Byte;
  var Byte2: Byte):integer;
```

Prototypes in VisualBasic

```
eProDas_DAC7611_BuildValue_Separated
```

The `Byte1` is the proper value to be consumed in the first part of the transfer whereas `Byte2` should be read in the second part.

# 21 Working with multiple devices simultaneously

eProDas device stack can work with several devices simultaneously, as the chapter 10 describes from the users' point of view. The maximal number of allowed devices is 16. Additional connections are refused by eProDas device driver and the devices that exceed the stated quota are not visible to the `eProDas.DLL` and consequently to your applications.

The number of permitted devices may be changed in the future so we do not recommend you to hardwire the number 16 into your applications. Instead, use the following function to learn about the maximal number of allowed simultaneously connected devices that your eProDas system supports.

### 21.1.1 GetMaxNumberOfDevices

Prototype in C/C++

```
unsigned int eProDas_GetMaxNumberOfDevices();
```

Prototype in Delphi

```
function eProDas_GetMaxNumberOfDevices:LongWord;
```

Prototype in VisualBasic

```
Function eProDas_GetMaxNumberOfDevices() As UInteger
```

The function does not need to report any errors so the requested number (16) is reported through the return value.

A bit more interesting to know is the actual number of connected devices, which is reported by the following function.

### 21.1.2 GetNumberOfDevices

Prototype in C/C++

```
unsigned int eProDas_GetNumberOfDevices();
```

Prototype in Delphi

```
function eProDas_GetNumberOfDevices:LongWord;
```

Prototype in VisualBasic

```
Function eProDas_GetNumberOfDevices() As UInteger
```

Again, there are no errors to report so the number of discovered devices (between 0 and 16) is reported through a return value (of course, the value 0 is returned when there are no connected devices).

Let us presume that we have just connected 3 devices to our host PC. If we call the function `GetNumberOfDevices` immediately after, the value 0 would be returned to us instead of the expected 3. This is due to the fact that eProDas refreshes its device list **only** when instructed to do so by calling the function `OpenHandle` (section 14.3.1). It is the task of `OpenHandle` to discover all connected devices and initialize them. The function also queries each device for its `DeviceType` (section 10.1) and `DeviceTag` (section 10.3) to build internal table of devices for later enumeration by application.

Whenever your application wants to refresh the list of devices (for example, if it suspects that some device has been disconnected by the user) it does it so by calling the function `OpenHandle` again. So, if the number of reported devices is now 3 but you disconnect all of them, the system will still report the number 3 until the next call to `OpenHandle`.

Such semantic has been implemented deliberately because it is much easier to develop applications if the application dictates when the list of devices is updated. Otherwise, the application would need to check the state of hardware before each eProDas API function call that communicates with devices. Of course, if device is disconnected, the error `eProDas_Error_WriteToUSBFailed` or something else would be returned when sending commands to it. The point is that although the device is physically disconnected it is still logically present until the application is ready to acknowledge the truth.

If the previous paragraph seems complicated, do not worry. In the majority of situations the application simply calls `OpenHandle` at the beginning of its execution. Then it checks that all the required devices are present. Then it simply proceeds under the presumption that such state will last till the end of the execution. If the user is awkward or malicious and disconnects the device(s), then the experiment will simply fail and that is okay. The exception may be the situations where eProDas devices control some equipment that could make injuries or material damage (like robotic arms) in which case the up to date state of connected devices may be refreshed by calls to the function `OpenHandle` at an appropriate moments of time.

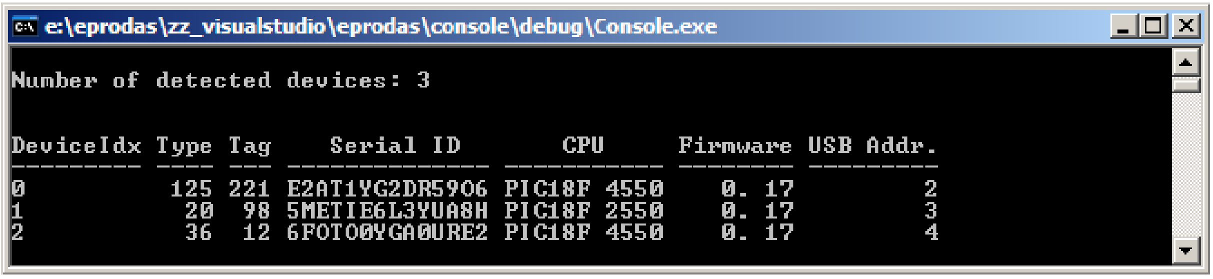
## 21.2 Device indexes

Undoubtedly, you know by now that all eProDas API functions for influencing the hardware posses the parameter `DeviceIdx`. This parameter enables us to specify the device that is the target of a certain command or action. Device indexes are zero based, so when there is only one connected device in the system, you always access it by specifying the zero value of `DeviceIdx`. In the case of three connected devices, the valid device indexes are 0, 1 and 2.

**Important!** Never assume any particular rule about how the physical devices and device indexes are mapped together. Always query each discovered device for its `DeviceType` (section 10.1) and/or `DeviceTag` (section 10.3) to learn which device index is mapped to a certain physical device.

For example, let us say that currently there are no connected devices to our eProDas system. Now we manually plug in a device with `DeviceType` of 125 and `DeviceTag` of 221. Since this is the first eProDas device in the system the eProDas device driver does not have much choice but to assign the first device object to it. Consequently, this device is accessed by specifying `DeviceIdx` of zero. Now let us plug in the second device with `DeviceType` of 20 and `DeviceTag` of 98. Chronologically, this is the second device so device driver assigns the second device object to it which links this device to `DeviceIdx` of one. The last one of our devices is the one with `DeviceType` of 36 and `DeviceTag` of 12, which is naturally accessed through `DeviceIdx` of two.

The console demo application (chapter 8) can be useful to check or diagnose our setup since it can list all connected devices, as the following figure reveals.



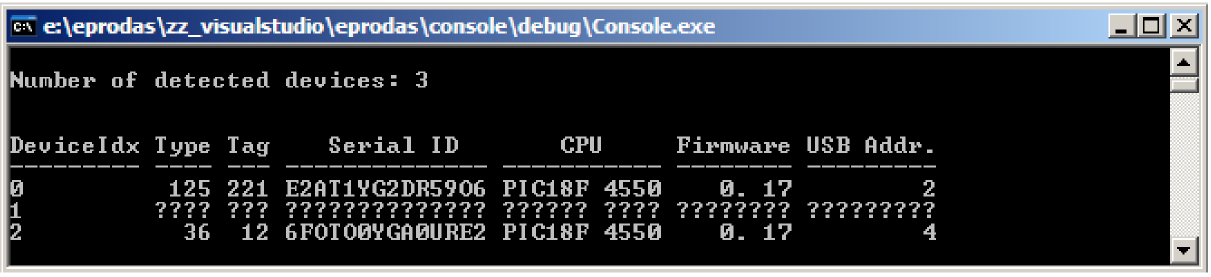
e:\eprodas\zz_visualstudio\eprodas\console\debug\Console.exe

```
Number of detected devices: 3

DeviceIdx Type Tag    Serial ID       CPU     Firmware USB Addr.
--------- ---- --- -------------- ----------- -------- ---------
0          125 221 E2AT1YG2DR5906 PIC18F 4550    0. 17         2
1           20  98 5METIE6L3YUA8H PIC18F 2550    0. 17         3
2           36  12 6FOTO0YGA0URE2 PIC18F 4550    0. 17         4
```

**Figure 94: List of connected eProDas devices.**

The first column lists device indexes, whereas the second and the third ones reveal `DeviceType` and `DeviceTag` of the respective devices. The rest of the information is not relevant at this moment.

Okay, let us deliberately disconnect the second device on the list and generate the listing again. The result is presented in the following figure.



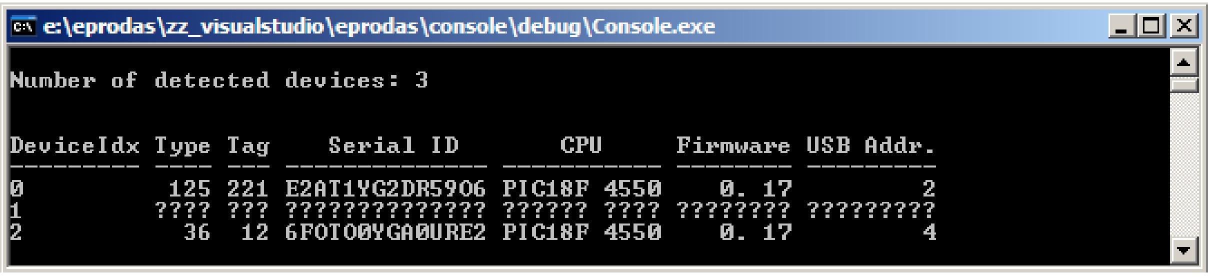
```
e:\eprodas\zz_visualstudio\eprodas\console\debug\Console.exe

Number of detected devices: 3

DeviceIdx Type Tag    Serial ID        CPU      Firmware USB Addr.
--------- ---- --- -------------- ----------- -------- ---------
0          125 221 E2AT1YG2DR59O6 PIC18F 4550    0. 17         2
1         ???? ??? ?????????????? ?????? ???? ???????? ?????????
2           36  12 6FOTO0YGA0URE2 PIC18F 4550    0. 17         4
```

**Figure 95: List of connected devices in the case of one unplugged device.**

As you can see, the system still reports 3 connected devices, as the first line of the output states. However, the second device on the list is not physically connected any more so the system is unable to query its data; console demo application in such cases prints out question marks instead of the data.

The important observation is that the system does not automatically rearrange device indexes when some device is disconnected (or connected). Instead, it does nothing at all to give you the guarantee that the remaining devices are still accessible by specifying the same `DeviceIdx` values as if the second device was not disconnected.

Okay, let us connect the missing device back to the same USB port. The list of devices is as follows.



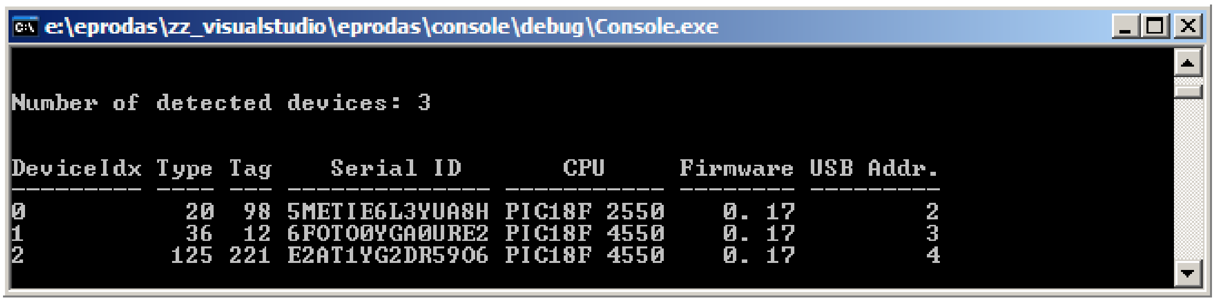
```
e:\eprodas\zz_visualstudio\eprodas\console\debug\Console.exe

Number of detected devices: 3

DeviceIdx Type Tag    Serial ID        CPU      Firmware USB Addr.
--------- ---- --- -------------- ----------- -------- ---------
0          125 221 E2AT1YG2DR59O6 PIC18F 4550    0. 17         2
1         ???? ??? ?????????????? ?????? ???? ???????? ?????????
2           36  12 6FOTO0YGA0URE2 PIC18F 4550    0. 17         4
```

**Figure 96: List of connected devices when the device is re-plugged back into the slot.**

The list remains completely unchanged, which leads to another important conclusion. If you unplug the device even only for a fraction of a second and then you plug it back in even in the same USB port, this is logically not the same device any more and you cannot access it until it is properly discovered by calling the function `OpenHandle` for one more time.

The reason for such semantic is obvious. When the device is disconnected it is disrupter and possibly reset. Its configuration is therefore altered so even if the system treats it as the same device, such device would not execute future actions in the same way as expected. It is far better for the application to be alerted about the hardware change than to keep it blind with potentially disastrous consequences in some scenarios (like malfunctioning robot's arm control).

So, let us now execute `OpenHandle` to acknowledge the reality and generate listing again with the following result.



e:\eprodas\zz_visualstudio\eprodas\console\debug\Console.exe

```
Number of detected devices: 3

DeviceIdx Type Tag     Serial ID        CPU      Firmware USB Addr.
--------- ---- --- --------------- ----------- -------- ---------
0           20  98 5METIE6L3YUA8H  PIC18F 2550    0. 17         2
1           36  12 6FOTO0YGA0URE2  PIC18F 4550    0. 17         3
2          125 221 E2AT1YG2DR5906  PIC18F 4550    0. 17         4
```

**Figure 97: Updated device list.**

As you can see, our three devices are back. However, note that their associated device indexes are not the same as before (in the Figure 94). Even the indexes of the two devices that were not disconnected during this time, have changed in the process.

Here comes the conclusion of the utmost importance. As soon as you call the function `OpenHandle`, you must NOT assume that ANY device index remains the same. Instead, you MUST query the mapping of EACH device index to find out which physical device is associated with it.

The best and the easiest way to assure such behaviour of your application is to create a special function, which opens the handle and then enumerates the discovered devices. Both processes must appear as an atomic operation to the surrounding application. Whenever you need to update the hardware information, simply call such function and everything will be done properly.

## 21.3 Enumeration of devices

By enumeration we mean the process of discovering which device index belongs to some physical device. Enumeration is based on querying the `DeviceType` and `DeviceTag` of each device with the following two functions.

### 21.3.1 GetDeviceType

Prototype in C/C++

```
int eProDas_GetDeviceType(unsigned int DeviceIdx, unsigned int &DeviceType);
```

Prototype in Delphi

```
function eProDas_GetDeviceType(DeviceIdx: LongWord;
  var DeviceType: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_GetDeviceType (ByVal DeviceIdx As UInteger,
  ByRef DeviceType As UInteger) As Integer
```

This function queries the eProDas device about its specific `DeviceType`, which is (of course) returned through the function parameter `DeviceType`.

### 21.3.2 GetDeviceTag

Prototype in C/C++

```
int eProDas_GetDeviceTag(unsigned int DeviceIdx, unsigned int &DeviceTag);
```

Prototype in Delphi

```
function eProDas_GetDeviceTag(DeviceIdx: LongWord;
  var DeviceTag: LongWord):integer;
```

Prototype in VisualBasic

```
Function eProDas_GetDeviceTag (ByVal DeviceIdx As UInteger,
  ByRef DeviceTag As UInteger) As Integer
```

Similarly, this function queries the eProDas device about its specific `DeviceTag`, which is returned through the function parameter `DeviceTag`.

So, all your application needs to do is to check the two non-volatile parameters for each present device to learn about its actual role in your setup.

eProDas makes the enumeration process even easier by providing the function `SearchDevice`, which you should prefer for doing enumeration instead of the previous two functions.

### 21.3.3 SearchDevice

Prototype in C/C++

```
int eProDas_SearchDevice(int Type, int Tag);
```

Prototype in Delphi

```
function eProDas_SearchDevice(DeviceType: integer; DeviceTag: integer):integer;
```

Prototype in VisualBasic

```
Function eProDas_SearchDevice (ByVal DeviceType As Integer,
  ByVal DeviceTag As Integer) As Integer
```

This function searches all (logically) connected devices and returns device index of the first one that fulfills the search criterion regarding demanded `DeviceType` and `DeviceTag`. If the specified device cannot be found, the value –1 is returned.

Let us go back to our previous exemplary hardware setup in the Figure 97. To discover which device index is associated with the device in the second row of the listing, we would call the function `SearchDevice` with parameters `DeviceType` of 36 and `DeviceTag` of 12. In this case the function would return the value of 1 to us. To learn about the whole setup a similar query would be done for each expected device.

If you want to take only one parameter into account when doing the quest for device index, you can set the irrelevant parameter to the value –1. So, to find a device with `DeviceTag` of 221 regardless of its `DeviceType`, execute the function call `eProDas_SearchDevice`(–1, 221). Similarly, to find a device with `DeviceType` of 20 regardless of its `DeviceTag`, execute `eProDas_SearchDevice`(20, –1). The function call `eProDas_SearchDevice`(–1, –1) returns any present device.

**Note 1.** If more than one device fulfills the criterion, there is no guarantee, which one of them will be returned. Such setup reflects a bad or ignorant configuration of individual `DeviceType` and `DeviceTag` parameters of your devices.

**Note 2.** Functions `GetDeviceType` and `GetDeviceTag` always query the up to date state of the device, so they are a bit inefficient since true USB data exchange takes place upon each function call. This also means that if the device is unplugged, the information cannot be retrieved. Contrary, the function `SearchDevice` never queries the physical device but it instead searches internal device table, which the function `OpenHandle` builds. No USB packets are exchanged with the device but hardware changes are also not reflected by results of the search. Usually, this is what your application wants.

### 21.3.4 Exemplar enumeration

Let as make an exemplar enumeration to see how easy it is to make the whole thing properly despite the complicated explanation. We shall start with the hardware setup in the Figure 97. To make the example more concrete we are now going to assign some descriptive roles to the three devices. Device with type 20 and tag 98 is used for voltage measurements whereas device with type 36 and tag 12 measures temperature. Finally, device with type 125 and tag 221 controls robot's arm.

The following code shows how to properly open the handle and enumerate the devices in our case.

```
 1.  unsigned int VoltageIdx, TemperatureIdx, RobotIdx;
 2.  int TempIdx;

 3.  int Status = eProDas_OpenHandle();
 4.  if(Status != eProDas_Error_SUCCESS) {
 5.    Abort_OpenHandle_Failed();
     }

 6.  TempIdx = eProDas_SearchDevice(20, 98);
 7.  if(TempIdx == -1) {
 8.    Abort_Requested_Device_Missing();
     }
 9.  VoltageIdx = TempIdx;

10.  TempIdx = eProDas_SearchDevice(36, 12);
11.  if(TempIdx == -1) {
12.    Abort_Requested_Device_Missing();
     }
13.  TemperatureIdx = TempIdx;

14.  TempIdx = eProDas_SearchDevice(125, 221);
15.  if(TempIdx == -1) {
16.    Abort_Requested_Device_Missing();
     }
17.  RobotIdx = TempIdx;
```

**Figure 98: Exemplar enumeration code.**

Line 1 declares three variables to hold the yet to be discovered device indexes. Line 2 declares temporal storage for index during enumeration. This is a bit of a complication which is due to the fact that device indexes are unsigned integers but the function `SearchDevice` returns negative result in the case of failure, so the discovered index needs to be stored in a signed temporal variable.

Line 3 tries to open the handle; lines 4 and 5 take care of aborting the experiment if the attempt fails.

Line 6 tries to obtain the index of the device for voltage measurements by using the proper search criterion. If no searched for device is connected to the system then lines 7 and 8 abort the experiment. In the opposite case in the line 9 the discovered index is stored into the appropriate (probably global) variable. You compiler may warn you about the signed/unsigned mismatch, which you can eliminate by an explicit type cast.

The logic for discovering the other two devices is completely the same. The bottom line is that the variables `VoltageIdx`, `TemperatureIdx` and `RobotIdx` always hold the indexes of the respective devices regardless of index mappings.

So, to write some value to some port of robot's arm controlling device, use the following function call: `Status = eProDas_WritePort(RobotIdx, SomePort, SomeValue)`. Similarly, the following function call is appropriate to read the currently acquired AD channel of voltage measurement device: `Status = eProDas_ReadInternalADChannel(VoltageIdx, Result)`.

# 22 Retrieving the entire configuration of the device

So far we have described many settings and options that are under the control of the user (programmer) of the eProDas device. All these settings, except the ones that fall within the scope of periodic actions, can be retrieved by the function `GetConfiguration`. Strictly speaking you do not need this function at all. Simply configure the device according to the requirements of your application and that's it. Still, in some cases you may want to examine the resulting configuration of the device, which is summarized by the function `GetConfiguration`.

The retrieved configuration is the actual one. This means that true PIC registers are examined to determine the real device state. If you modify configuration through the usage of the developers' set of function or if you directly write to PIC configuration registers by any means (like with In-Circuit debugger) the retrieved configuration will reflect all that backdoor changes and not just settings that are done by the functions that belong to eProDas API.

## 22.1 GetConfiguration

Prototype in C/C++

```
int eProDas_GetConfiguration(unsigned int DeviceIdx,
  eProDasStruct_Configuration &ConfigInfo);
```

Prototype in Delphi

```
function eProDas_GetConfiguration(DeviceIdx: LongWord;
  var ConfigInfo: TeProDasStruct_Configuration):integer;
```

Prototype in VisualBasic

```
Function eProDas_GetConfiguration (ByVal DeviceIdx As UInteger,
  ByRef ConfigInfo As TeProDasStruct_Configuration) As Integer
```

The definition of the function looks rather simple. Besides device index there is only parameter that specifies a placeholder for the result. However, this second parameter is a rather elaborate structure with many fields. When additional features of device are implemented this structure will expand to cover all of them. The definition of the structure in all incarnations can be examined in the following figures.

```
struct eProDasStruct_Configuration {
  unsigned int Errors; //0 = no error

  unsigned int Ports[5*8]; //0..7 = PortA,
  8..15 = PortB, 16..23 = PortC, 24..31 = PortD, 32..39 = PortE

  unsigned int *PortA() { return Ports+0; }
  unsigned int *PortB() { return Ports+8; }
  unsigned int *PortC() { return Ports+16; }
  unsigned int *PortD() { return Ports+24; }
  unsigned int *PortE() { return Ports+32; }
  unsigned int InternalPullUpsEnabled;

  unsigned int NumberOfInternalADChannels; //0 = AD module is off
  unsigned int SelectedInternalADChannel; //0 = AD module is off
  unsigned int InternalADRefPlus; //0 = VDD, 1 = VrefPlus pin
  unsigned int InternalADRefMinus; //0 = VSS, 1 = VrefMinus pin
  unsigned int InternalADJustify; //0 = left justify, 1 = right justify
  unsigned int InternalADClock; //Hz
  unsigned int InternalADAcquisitionCode; //from 0 to 7
  unsigned int InternalADAcquisitionClocks; //Tad
  unsigned int InternalADAcquisitionTime; //micros

  unsigned int InternalComparatorsOn;
  unsigned int InternalComparatorsMode;
  unsigned int InternalComparator1On;
  unsigned int InternalComparator2On;
  unsigned int InternalComparator1Inv;
  unsigned int InternalComparator2Inv;
  unsigned int InternalComparatorInputSwitch;

  unsigned int InternalReferenceEnabled;
  unsigned int InternalReferenceValue;
  unsigned int InternalReferenceRange;
  unsigned int InternalReferenceSourceOnPin;
  unsigned int InternalReferenceOutputOnPin;

  unsigned int InternalTimer1On;
  unsigned int InternalTimer1Value;
  unsigned int InternalTimer1ExternalClockSource;
  unsigned int InternalTimer1ExternalClockSync;
  unsigned int InternalTimer1OscillatorEnable;
  unsigned int InternalTimer1Prescaler;
  unsigned int InternalTimer1IsDeviceClock;
  unsigned int InternalTimer1ReadWrite16Bit;

  unsigned int InternalTimer2On;
  unsigned int InternalTimer2Value;
  unsigned int InternalTimer2PeriodRegister;
  unsigned int InternalTimer2Prescaler;
  unsigned int InternalTimer2Postscaler;
  unsigned int InternalTimer2PeriodClocks;

  unsigned int InternalTimer3On;
  unsigned int InternalTimer3Value;
  unsigned int InternalTimer3ExternalClockSource;
  unsigned int InternalTimer3ExternalClockSync;
  unsigned int InternalTimer3Prescaler;
  unsigned int InternalTimer3ReadWrite16Bit;
```

**Figure 99: The C/C++ definition of the `eProDasStruct_Configuration` (part 1).**

```
    unsigned int InternalPWM1On;
    unsigned int InternalPWM1PeriodClocks;
    unsigned int InternalPWM1DutyCycleClocks;
    unsigned int InternalPWM1Mode;
    unsigned int InternalPWM1ActiveStateP1A;
    unsigned int InternalPWM1ActiveStateP1B;
    unsigned int InternalPWM1ActiveStateP1C;
    unsigned int InternalPWM1ActiveStateP1D;
    unsigned int InternalPWM1HalfBridgeDelay;
    unsigned int AutoShutDownSource;
    unsigned int AutoShutDownState;
    unsigned int InternalPWM1AutoShutdownStateP1A;
    unsigned int InternalPWM1AutoShutdownStateP1B;
    unsigned int InternalPWM1AutoShutdownStateP1C;
    unsigned int InternalPWM1AutoShutdownStateP1D;
    unsigned int InternalPWM1AutoRestart;

    unsigned int InternalPWM2On;
    unsigned int InternalPWM2PeriodClocks;
    unsigned int InternalPWM2DutyCycleClocks;
    unsigned int InternalPWM2MuxPin; //RC1 = 1, RB3 = 0

    unsigned int InternalSPIOn;
    unsigned int InternalSPIMode;
    unsigned int InternalSPI_IN_SampleAtTheEnd;
    unsigned int InternalSPITransmitOnActiveToIdle;
    unsigned int InternalSPIClockIdleStateHigh;
    unsigned int InternalSPIOutputDemanded;
    unsigned int InternalSPIInputDemanded;

    unsigned int InternalUSARTOn;
    unsigned int InternalUSARTBaudRate;
    unsigned int InternalUSARTTransmitEnabled;
    unsigned int InternalUSARTOutputDemanded;
    unsigned int InternalUSARTInputDemanded;

};
```

**Figure 100: The C/C++ definition of the `eProDasStruct_Configuration` (part 2).**

```
TeProDasStruct_Configuration = record
  Errors: LongWord; //0 = no error

  Ports: array[0..39] of LongWord; //0..7 = PortA,
  8..15 = PortB, 16..23 = PortC, 24..31 = PortD, 32..39 = PortE
  InternalPullUpsEnabled: LongWord;

  NumberOfInternalADChannels: LongWord; //0 = AD module is off
  SelectedInternalADChannel: LongWord; //0 = AD module is off
  InternalADRefPlus: LongWord; //0 = VDD, 1 = VrefPlus pin
  InternalADRefMinus: LongWord; //0 = VSS, 1 = VrefMinus pin
  InternalADJustify: LongWord; //0 = left justify, 1 = right justify
  InternalADClock: LongWord; //Hz
  InternalADAcquisitionCode: LongWord; //from 0 to 7
  InternalADAcquisitionClocks: LongWord; //Tad
  InternalADAcquisitionTime: LongWord; //micros

  InternalComparatorsOn: LongWord;
  InternalComparatorsMode: LongWord;
  InternalComparator1On: LongWord;
  InternalComparator2On: LongWord;
  InternalComparator1Inv: LongWord;
  InternalComparator2Inv: LongWord;
  InternalComparatorInputSwitch: LongWord;

  InternalReferenceEnabled: LongWord;
  InternalReferenceValue: LongWord;
  InternalReferenceRange: LongWord;
  InternalReferenceSourceOnPin: LongWord;
  InternalReferenceOutputOnPin: LongWord;

  InternalTimer1On: LongWord;
  InternalTimer1Value: LongWord;
  InternalTimer1ExternalClockSource: LongWord;
  InternalTimer1ExternalClockSync: LongWord;
  InternalTimer1OscillatorEnable: LongWord;
  InternalTimer1Prescaler: LongWord;
  InternalTimer1IsDeviceClock: LongWord;
  InternalTimer1ReadWrite16Bit: LongWord;

  InternalTimer2On: LongWord;
  InternalTimer2Value: LongWord;
  InternalTimer2PeriodRegister: LongWord;
  InternalTimer2Prescaler: LongWord;
  InternalTimer2Postscaler: LongWord;
  InternalTimer2PeriodClocks: LongWord;

  InternalTimer3On: LongWord;
  InternalTimer3Value: LongWord;
  InternalTimer3ExternalClockSource: LongWord;
  InternalTimer3ExternalClockSync: LongWord;
  InternalTimer3Prescaler: LongWord;
  InternalTimer3ReadWrite16Bit: LongWord;
```

**Figure 101: The Delphi definition of the `eProDasStruct_Configuration` (part 1).**

```
    InternalPWM1On: LongWord;
    InternalPWM1PeriodClocks: LongWord;
    InternalPWM1DutyCycleClocks: LongWord;
    InternalPWM1Mode: LongWord;
    InternalPWM1ActiveStateP1A: LongWord;
    InternalPWM1ActiveStateP1B: LongWord;
    InternalPWM1ActiveStateP1C: LongWord;
    InternalPWM1ActiveStateP1D: LongWord;
    InternalPWM1HalfBridgeDelay: LongWord;
    AutoShutDownSource: LongWord;
    AutoShutDownState: LongWord;
    InternalPWM1AutoShutdownStateP1A: LongWord;
    InternalPWM1AutoShutdownStateP1B: LongWord;
    InternalPWM1AutoShutdownStateP1C: LongWord;
    InternalPWM1AutoShutdownStateP1D: LongWord;
    InternalPWM1AutoRestart: LongWord;

    InternalPWM2On: LongWord;
    InternalPWM2PeriodClocks: LongWord;
    InternalPWM2DutyCycleClocks: LongWord;
    InternalPWM2MuxPin: LongWord; //RC1 = 1, RB3 = 0

    InternalSPIOn: LongWord;
    InternalSPIMode: LongWord;
    InternalSPI_IN_SampleAtTheEnd: LongWord;
    InternalSPITransmitOnActiveToIdle: LongWord;
    InternalSPIClockIdleStateHigh: LongWord;
    InternalSPIOutputDemanded: LongWord;
    InternalSPIInputDemanded: LongWord;

    InternalUSARTOn: LongWord;
    InternalUSARTBaudRate: LongWord;
    InternalUSARTTransmitEnabled: LongWord;
    InternalUSARTOutputDemanded: LongWord;
    InternalUSARTInputDemanded: LongWord;

end;
```

**Figure 102: The Delphi definition of the `eProDasStruct_Configuration` (part 2).**

```
Public Structure TeProDasStruct_Configuration
    Public Errors As UInteger ' 0 is no errors

    Public Pin_RA0 As UInteger
    Public Pin_RA1 As UInteger
    Public Pin_RA2 As UInteger
    Public Pin_RA3 As UInteger
    Public Pin_RA4 As UInteger
    Public Pin_RA5 As UInteger
    Public Pin_RA6 As UInteger
    Public Pin_RA7 As UInteger

    Public Pin_RB0 As UInteger
    Public Pin_RB1 As UInteger
    Public Pin_RB2 As UInteger
    Public Pin_RB3 As UInteger
    Public Pin_RB4 As UInteger
    Public Pin_RB5 As UInteger
    Public Pin_RB6 As UInteger
    Public Pin_RB7 As UInteger

    Public Pin_RC0 As UInteger
    Public Pin_RC1 As UInteger
    Public Pin_RC2 As UInteger
    Public Pin_RC3 As UInteger
    Public Pin_RC4 As UInteger
    Public Pin_RC5 As UInteger
    Public Pin_RC6 As UInteger
    Public Pin_RC7 As UInteger

    Public Pin_RD0 As UInteger
    Public Pin_RD1 As UInteger
    Public Pin_RD2 As UInteger
    Public Pin_RD3 As UInteger
    Public Pin_RD4 As UInteger
    Public Pin_RD5 As UInteger
    Public Pin_RD6 As UInteger
    Public Pin_RD7 As UInteger

    Public Pin_RE0 As UInteger
    Public Pin_RE1 As UInteger
    Public Pin_RE2 As UInteger
    Public Pin_RE3 As UInteger
    Public Pin_RE4 As UInteger
    Public Pin_RE5 As UInteger
    Public Pin_RE6 As UInteger
    Public Pin_RE7 As UInteger

    Public InternalPullUpsEnabled As UInteger
```

**Figure 103: The VisualBasic definition of the `eProDasStruct_Configuration` (part 1).**

```
    Public NumberOfInternalADChannels As UInteger
    Public SelectedInternalADChannel As UInteger
    Public InternalADRefPlus As UInteger
    Public InternalADRefMinus As UInteger
    Public InternalADJustify As UInteger
    Public InternalADClock As UInteger
    Public InternalADAcquisitionCode As UInteger
    Public InternalADAcquisitionClocks As UInteger
    Public InternalADAcquisitionTime As UInteger

    Public InternalComparatorsOn As UInteger
    Public InternalComparatorsMode As UInteger
    Public InternalComparator1On As UInteger
    Public InternalComparator2On As UInteger
    Public InternalComparator1Inv As UInteger
    Public InternalComparator2Inv As UInteger
    Public InternalComparatorInputSwitch As UInteger

    Public InternalReferenceEnabled As UInteger
    Public InternalReferenceValue As UInteger
    Public InternalReferenceRange As UInteger
    Public InternalReferenceSourceOnPin As UInteger
    Public InternalReferenceOutputOnPin As UInteger

    Public InternalTimer1On As UInteger
    Public InternalTimer1Value As UInteger
    Public InternalTimer1ExternalClockSource As UInteger
    Public InternalTimer1ExternalClockSync As UInteger
    Public InternalTimer1OscillatorEnable As UInteger
    Public InternalTimer1Prescaler As UInteger
    Public InternalTimer1IsDeviceClock As UInteger
    Public InternalTimer1ReadWrite16Bit As UInteger

    Public InternalTimer2On As UInteger
    Public InternalTimer2Value As UInteger
    Public InternalTimer2PeriodRegister As UInteger
    Public InternalTimer2Prescaler As UInteger
    Public InternalTimer2Postscaler As UInteger
    Public InternalTimer2PeriodClocks As UInteger

    Public InternalTimer3On As UInteger
    Public InternalTimer3Value As UInteger
    Public InternalTimer3ExternalClockSource As UInteger
    Public InternalTimer3ExternalClockSync As UInteger
    Public InternalTimer3Prescaler As UInteger
    Public InternalTimer3ReadWrite16Bit As UInteger
```

**Figure 104: The VisualBasic definition of the `eProDasStruct_Configuration` (part 2).**

```
    Public InternalPWM1On As UInteger
    Public InternalPWM1PeriodClocks As UInteger
    Public InternalPWM1DutyCycleClocks As UInteger
    Public InternalPWM1Mode As UInteger
    Public InternalPWM1ActiveStateP1A As UInteger
    Public InternalPWM1ActiveStateP1B As UInteger
    Public InternalPWM1ActiveStateP1C As UInteger
    Public InternalPWM1ActiveStateP1D As UInteger
    Public InternalPWM1HalfBridgeDelay As UInteger
    Public AutoShutDownSource As UInteger
    Public AutoShutDownState As UInteger
    Public InternalPWM1AutoShutdownStateP1A As UInteger
    Public InternalPWM1AutoShutdownStateP1B As UInteger
    Public InternalPWM1AutoShutdownStateP1C As UInteger
    Public InternalPWM1AutoShutdownStateP1D As UInteger
    Public InternalPWM1AutoRestart As UInteger

    Public InternalPWM2On As UInteger
    Public InternalPWM2PeriodClocks As UInteger
    Public InternalPWM2DutyCycleClocks As UInteger
    Public InternalPWM2MuxPin As UInteger ' //RC1 = 1, RB3 = 0

    Public InternalSPIOn As UInteger
    Public InternalSPIMode As UInteger
    Public InternalSPI_IN_SampleAtTheEnd As UInteger
    Public InternalSPITransmitOnActiveToIdle As UInteger
    Public InternalSPIClockIdleStateHigh As UInteger
    Public InternalSPIOutputDemanded As UInteger
    Public InternalSPIInputDemanded As UInteger

    Public InternalUSARTOn As UInteger
    Public InternalUSARTBaudRate As UInteger
    Public InternalUSARTTransmitEnabled As UInteger
    Public InternalUSARTOutputDemanded As UInteger
    Public InternalUSARTInputDemanded As UInteger

End Structure
```

**Figure 105: The VisualBasic definition of the `eProDasStruct_Configuration` (part 3).**

## 22.2 Description of fields in the configuration record

The bit field `Errors` summarizes any discovered errors or inconsistencies of the configuration. If you configured the device solely by using the users' set of functions there should be no errors at all (except the selected AD channel may be greater than the number of AD channels, see section 15.2.3). Contrary to this, other means of doing the configuration may lead to many errors, like an analog input pin being configured as digital output by means of direct writing to PIC registers.

The names of bits in `Errors` filed are summarized in the Table 21.

| Name | Value |
|---|---:|
| `eProDas_ConfigError_Pins` | 1 |
| `eProDas_ConfigError_ADChannels` | 2 |
| `eProDas_ConfigError_ADChannel` | 4 |
| `eProDas_ConfigError_Reference` | 8 |

**Table 21: The names of `Errors` filed in the configuration record.**

- Bit `eProDas_ConfigError_Pins` indicates whether any errors were discovered in pins' configuration. Which pins are configured in a wrong way is further specified by the `Ports` array (by the entries `Pin_Rxy` in the case of Visual Basic) of the configuration record.
- Non-zero value of bit `eProDas_ConfigError_ADChannels` means that configured number of AD channels is greater than 13, although there are only 13 physically implemented AD inputs.
- Non-zero value of bit `eProDas_ConfigError_ADChannel` means that currently selected AD channel is not configured to be AD input. For example, you configure AD module to have only one AD input, but channel AN2 is currently selected.
- Bit `eProDas_ConfigError_Reference` indicates that both the source and the output of voltage reference module are connected to the pin RA2 at the same time.

### 22.2.1 Configuration of individual pins

The `Ports` field is an array of forty 32-bit fields (in the case of Visual Basic there is a separate scalar entry for each pin), where each bit filed summarizes the role of one PIC's pin. Entries from 0 to 7 belong to `PORTA` pins, 8 to 15 are associated with `PORTB`, etc. There are exactly 8 entries for each port regardless of the number of implemented pins. This organization (hopefully) makes the navigation of the array easier.

The name of individual bits of each `Ports` entry is summarized in the Table 22. Each pin's bit field may have more than one bit set if the pin fulfils more than one role, so generally you must not compare the value of the `Ports` field against the entries in the Table 22 but you must instead bitmask the entries of interest.

| Name | Value |
|---|---:|
| eProDas_Pin_Error | 1 |
| eProDas_Pin_Unimplemented | 2 |
| eProDas_Pin_Reserved | 4 |
| eProDas_Pin_DigitalOut | 8 |
| eProDas_Pin_Analog | 16 |
| eProDas_Pin_ADInput | 32 |
| eProDas_Pin_ADPlus | 64 |
| eProDas_Pin_ADMinus | 128 |
| eProDas_Pin_CompPlus | 256 |
| eProDas_Pin_CompMinus | 512 |
| eProDas_Pin_CompOut | 1024 |
| eProDas_Pin_RefPlus | 2048 |
| eProDas_Pin_RefMinus | 4096 |
| eProDas_Pin_RefOut | 8192 |
| eProDas_Pin_TimerIn | 16384 |
| eProDas_Pin_PWMOut | 32768 |
| eProDas_Pin_SPIOut | 65536 |
| eProDas_Pin_SPIIn | 131072 |
| eProDas_Pin_SPIClk | 262144 |
| eProDas_Pin_USARTOut | 524288 |
| eProDas_Pin_USARTIn | 1048576 |

**Table 22: The names of bits in the `Ports` array entry.**

- Bit `eProDas_Pin_Error` is set whenever at least one error or inconsistency is discovered in the configuration of an associated pin.

- Bit `eProDas_Pin_Unimplemented` is set if the pin does not physically exist, e.g. RC3.

- Bit `eProDas_Pin_Reserved` indicates that the pin exists, but it is not available as a general purpose pin, e.g. RA6.

- Bit `eProDas_Pin_DigitalOut` indicates that pin is configured as digital output.

- If pin has at least one analog function then bit `eProDas_Pin_Analog` is set. If this bit is cleared then pin has a pure digital role; the previous bit indicates whether it is digital input or output.

- If pin can be source for AD conversion the bit `eProDas_Pin_ADInput` is set.

- Bits `ADPlus` and `ADMinus` indicate that AD module gets plus or minus voltage reference through the pin, respectively. Only pins RA2 and RA3 can have the appropriate one of these bits set.

- Bits `CompPlus` and `CompMinus` indicate that pin is noninverting or inverting input, respectively, of the voltage comparator.

- Bit `CompOut` is set when the pin outputs the asynchronous comparator result.

- Bits `RefPlus` and `RefMinus` indicate that pin is plus or minus voltage reference, respectively, of the voltage reference module.

- Bit `RefOut` is set when the pin outputs the voltage of the voltage reference module.

- Bit `TimerIn` is set when the pin is a clock source for one of the timer modules.

- Bit `PWMOut` is set for the pin that is the output of a turned-on PWM module. In order to actually output the PWM signal the pin must be also configured as digital output.
- Bit `SPIOut` is set when the SPI module transmits data through the associated pin (RC7 only).
- Bit `SPIIn` is set when the SPI module receives data through the associated pin (RB0 only).
- Bit `SPIClk` is set when the pin outputs (in master mode) or inputs (in slave mode) SPI clock (RB1 only).
- Bit `USARTOut` is set when the USART module transmits data through the associated pin (RC6 only).
- Bit `USARTIn` is set when the USART module receives data through the associated pin (RC7 only).

Warning! The access of the `Ports` array is completely unchecked. This unfortunate situation is due to the fact that not all languages support private members and/or functions of plain structures. For example, in Delphi we would need to derive `TeProDasStruct_Configuration` structure from TObject type to enable the checked array access through methods. This would render the structure incompatible on a bitwise level with a C/C++ one without additional glue layer of code.

The problem could be solved by implementing additional wrapper functions and by kindly asking each programmer to use these instead of plain array access. In the perfect world it might work but not on this planet. Those programmers that would take extreme care when accessing arrays anyway would use the wrapper functions; the less cautious one would surely ignore them entirely.

The field `InternalPullUpsEnabled` holds a non-zero value if pull-up resistors of port B are enabled.

### 22.2.2 Configuration of internal AD module

As the name reveals the field `NumberOfInternalADChannels` holds the configured number of AD channels of the PIC's AD module. When this value is zero, the module is turned off.

The field `SelectedInternalADChannel` indicates the currently acquired AD channel, i.e. the channel that would be targeted by the function `ReadInternalADChannel` (see the Table 6 on page 78).

The fields `InternalADRefPlus` and `InternalADRefMinus` indicate whether the positive or negative AD reference rail, respectively, is connected to the power supply pin (the value of 0) or to the appropriate pin RA2 or RA3 (the value of 1).

When left justification of the AD result is selected the field `InternalADJustify` holds the value of 1; 0 otherwise (section 15.2.2).

The selected AD clock in [Hz] is contained in the field `InternalADClock`. As stated in the section 15.2.2, AD conversion takes 11 AD clocks to execute.

The field `InternalADAcquisitionCode` contains the selected AD acquisition time code (from 0 to 7; see the Table 7 on page 79). The field `InternalADAcquisitionClocks` reveals the associated acquisition period in number of AD clocks. Finally, the resulting acquisition time in [µs] can be obtained by reading the field `InternalADAcquisitionTime`.

### 22.2.3 Configuration of internal comparators

The field `InternalComparatorsOn` holds a non-zero value if the comparators' module is turned on and zero otherwise. The selected mode of comparators' module (from 0 to 7, see Figure 37 on page 84) can be obtained by reading the field `InternalComparatorsMode`.

According to the selected mode none, one or both comparators may be active, which is indicated by a non-zero value of the fields `InternalComparator1On` and `InternalComparator2On` for comparator one and two, respectively. If the comparator is inactive, the associated field has a zero value.

The input switch bit for mode 6 is held in the field `InternalComparatorInputSwitch`.

### 22.2.4 Configuration of internal reference voltage

The field `InternalReferenceEnabled` holds a non-zero value if voltage reference module is enabled.

The selected value (from 0 to 15) is held in the filed `InternalReferenceValue`. The high and low range is indicated by the non-zero and zero value of the field `InternalReferenceRange`, respectively.

When the reference voltage of the module is obtained by the power supply pins the value of zero is held in the field `InternalReferenceSourceOnPin`; non-zero value otherwise.

A non-zero value of the field `InternalReferenceOutputOnPin` indicates that the output voltage is available on the pin RA2; otherwise the value is zero.

### 22.2.5 Configuration of timer1

The field `InternalTimer1On` holds a non-zero value if module `Timer1` is enabled.

The current value of timer 1 is contained in the field `InternalTimer1Value`. Please note, if timer is not configured for 16 bit reads/writes the contained value may be wrong if an overflow of least significant byte of timer occurs during the reading of the value.

The field `InternalTimer1ExternalClockSource` holds a non-zero value if timer's 1 clock is obtained through pin RC1. Zero value indicates that clock source is the internal instruction cycle clock (oscillator clock/4; 12 MHz if processor runs at full speed PLL clock, which is the only choice that enables a working USB connection).

The field `InternalTimer1ExternalClockSync` holds a non-zero value when external clock source is synchronized with internal instruction clock.

The field `InternalTimer1OscillatorEnable` holds a non-zero value when amplifier for external 31 kHz quartz oscillator (pins RC0 and RC1) is active. Otherwise external clock is obtained by means of driving pin RC1 with an external clock source.

The value of field `InternalTimer1Prescaler` equals the prescaler division ratio of the module.

The field `InternalTimer1IsDeviceClock` holds a non-zero value if device clock is derived from timer's 1 oscillator source.

The field `InternalTimer1ReadWrite16Bit` holds a non-zero value if timer is read in one atomic 16 bit operation. This way the read value can never be wrong due to an overflow of least significant byte during the two separate readings of both registers. Similar functionality for writes parallels the one for reads.

### 22.2.6 Configuration of timer2

The field `InternalTimer2On` holds a non-zero value if module `Timer2` is enabled.

The current value of timer 2 is contained in the field `InternalTimer2Value`.

The value of field `InternalTimer2Prescaler` equals the prescaler division ratio of the module.

The value `InternalTimer2Postscaler` equals the postscaler division ratio of the module.

The value `InternalTimer2PeriodRegister` equals the value of period register. When the value of timer 2 matches the value in this field, the timer 2 is automatically reset on the next clock.

The value `InternalTimer2PeriodClocks` equals the period of timer 2 in clocks. The value calculates as `(InternalTimer2PeriodRegister+1)*InternalTimer2Prescaler`.

### 22.2.7 Configuration of timer3

The field `InternalTimer3On` holds a non-zero value if module `Timer3` is enabled.

The current value of timer 3 is contained in the field `InternalTimer3Value`. Please note, if timer is not configured for 16 bit reads/writes the contained value may be wrong if an overflow of least significant byte of timer occurred during reading the value.

The field `InternalTimer3ExternalClockSource` holds a non-zero value if timer's 3 clock is obtained through pin RC1. Zero value indicates that clock source is the internal instruction cycle clock.

The value of field `InternalTimer3Prescaler` equals the prescaler division ratio of the module.

The field `InternalTimer3ExternalClockSync` holds a non-zero value when external clock source is synchronized with internal instruction clock.

The field `InternalTimer3ReadWrite16Bit` holds a non-zero value if the timer is read in one atomic 16 bit operation (see the equivalent field of timer1 module).

### 22.2.8 Configuration of internal PWM modules

The fields `InternalPWM1On` and `InternalPWM2On` hold a non-zero value when the respective PWM modules are operational state.

The fields `InternalPWM1PeriodClocks` and `InternalPWM2PeriodClocks` both hold the same value, which further equals the value of the field `InternalTimer2PeriodClocks`. The value specifies the duration of PWM period in 12 MHz clock ticks.

The fields `InternalPWM1DutyCycleClocks` and `InternalPWM2DutyCycleClocks` hold the duration of respective PWM signals in active state. The duration is specified in 48 MHz clock ticks.

The field `InternalPWM1Mode` specifies mode of PWM1 operation. The numerical values are the same as for selecting mode of operation upon module configuration.

The fields from `InternalPWM1ActiveStateP1A` to `InternalPWM1ActiveStateP1D` reveal the digital state, which drives respective switches in conduction. Pins P1A and P1C always have the same active/passive state. The same rule holds for pins P1B and P1D.

`InternalPWM1HalfBridgeDelay` holds the value of configured delay that is applicable to half-bridge mode of operation.

By examining the field `AutoShutDownSource` you can learn about the configured sources that trigger auto shutdown condition. The numerical values are the same as for selecting auto shutdown source upon module configuration.

The non-zero value of the field `AutoShutDownState` indicates that module PWM1 is currently in auto shutdown condition.

The set of fields `InternalPWM1AutoShutdownStateP1x` reveal the auto shutdown state of their respective pins. Values of 0 and 1 hold the digital state whereas value of 2 means that the pin is held under the tri-state (high impedance) condition during the auto shutdown.

When the field `InternalPWM1AutoRestart` holds a non-zero value module PWM1 is configured to automatically exit shutdown state when shutdown condition ceases to exist.

By examining the field `InternalPWM2MuxPin` you can determine which pin outputs signal of module PWM2. It can be either RC1 (non-zero value of the field) or RB3 (zero value of the field).

### 22.2.9 Configuration of SPI module

The field `InternalSPIOn` holds a non-zero value when the SPI module is operational.

The field `InternalSPIMode` reveals the mode of SPI module operation according to the following table.

| Mode of operation | Value |
|---|---:|
| master mode with 12 MHz serial clock | 0 |
| master mode with 3 MHz serial clock | 1 |
| master mode with 750 kHz serial clock | 2 |
| master mode with serial clock by module timer2 (also divided by 2) | 3 |
| slave mode with pin SlaveSelect enabled | 4 |
| slave mode with pin SlaveSelect disabled | 5 |

**Table 23: Modes of internal SPI module operation.**

When in any of the master modes the field `InternalSPI_IN_SampleAtTheEnd` holds a non-zero value if the received bits are sampled at the clock transition that finishes the interval in which the asserted bit is valid; otherwise the signal is sampled in the middle of the associated interval.

When in any of the slave modes a non-zero value of the field `InternalSPITransmitOnActiveToIdle` indicates that PIC outputs the next bit of data on clock transition from active to idle state; otherwise the bit transition occurs on clock transition from idle to active state.

A non-zero value of the field `InternalSPIClockIdleStateHigh` indicates that clock is held in high state (logic 1) during its inactive (idle) periods; otherwise the idle state is low (logic 0).

A non-zero value of the fields `InternalSPIOutputDemanded` and `InternalSPIInputDemanded` indicates that upon configuration of the SPI module the user demanded SPI output through pin RC7 and SPI input through pin RB0. The former means that this module may content for the pin with the USART module, if the latter has enabled reception (which is indicated by a non-zero value of the field `InternalUSARTInputDemanded`).

### 22.2.10 Configuration of USART module

The field `InternalUSARTOn` holds a non-zero value when USART module is currently operational. If the module is configured but not enabled (see section 16.6.1), the value of the field is zero.

The field `InternalUSARTBaudRate` holds the currently configured baud rate.

The field `InternalUSARTTransmitEnabled` holds a non-zero value if transmission by USART module is enabled.

The fields `InternalUSARTOutputDemanded` and `InternalUSARTInputDemanded` hold a non-zero value if upon configuration of the USART module the user demanded USART output through pin RC6 and input through pin RC7, respectively. Non-zero `InternalUSARTInputDemanded` means that this module may content for the pin RC7 with the SPI module, if SPI module has enabled transmission (non-zero value of the field `InternalSPIOutputDemanded`).

# 23 Esoteric or rarely used functions

This chapter contains descriptions of functions that are not interested to a broad range of eProDas developers or they are tedious to work with. Nonetheless, in certain scenarios of usage these functions are invaluable and power developers are expected to find these functions fairly useful.

## 23.1 Operations on the whole eProDas device

### 23.1.1 SetDeviceType

Prototype in C/C++

```
int eProDas_SetDeviceType(unsigned int DeviceIdx, unsigned int DeviceType);
```

Prototype in Delphi

```
function eProDas_SetDeviceType(DeviceIdx: LongWord; DeviceType: LongWord):integer;
```

Prototype in VisualBasic

```
eProDas_SetDeviceType
```

This function writes an 8-bit `Device Type` of your choice into the data EEPROM of eProDas device, therefore assigns it hardware identification number (section 10.1). The parameter `DeviceTag` must contain value between 0 and 255, or the error `eProDas_Error_InvalidParameter` will result.

If write operation fails for any reason the error `eProDas_Error_WriteToDataEEPROMFailed` will result. This may be due to the malfunction of the on-chip EEPORM which has limited write/erase endurance. However, according to the PIC's datasheet it should be possible to change the value at least 100,000 times before the EEPROM gets destroyed. If you are not even remotely close to this number and you still get the stated error you may strongly suspect that there is a bug somewhere in the eProDas device stack.

Also note that, eProDas first checks whether the new type differs from the current one. If both types are equal no writing to data EEPROM occurs, which saves EEPROM memory. Therefore, it is not possible to burn your chip by calling the function `SetType` in a loop, if the value remains the same.

### 23.1.2 SetDeviceTag

Prototype in C/C++

```
int eProDas_SetDeviceTag(unsigned int DeviceIdx, unsigned int DeviceTag);
```

Prototype in Delphi

```
function eProDas_SetDeviceTag(DeviceIdx: LongWord; DeviceTag: LongWord):integer;
```

Prototype in VisualBasic

```
eProDas_SetDeviceTag
```

By calling this function your application assigns an 8-bit tag to a device. The parameter `DeviceTag`, which must contain value between 0 and 255, or the error `eProDas_Error_InvalidDeviceTag` will result.

Whenever using this function when more than one eProDas device is connected at once make sure that you know which one of them will be tagged. The mess that may arise by wrongly assigned tags is the reason for our recommendation to use the `SetTag` utility with only one device connected at the time.

The tag is written into the PIC’s data EEPROM. If write operation fails for any reason the error `eProDas_Error_WriteToDataEEPROMFailed` will result. This may be due to the malfunction of the on-chip EEPORM which has limited write/erase endurance. However, according to the PIC’s datasheet it should be possible to change the tag value at least 100,000 times before the EEPROM gets destroyed. If you are not even remotely close to this number and you still get the stated error you may strongly suspect that there is a bug somewhere in the eProDas device stack.

Also note that, eProDas first checks whether the new tag differs from the current one. If both tags are equal no writing to data EEPROM occurs, which saves EEPROM memory. Therefore, it is not possible to burn your chip by calling the function `SetTag` in a loop, if the tag value remains the same.

## 23.2 Transfer backbone

eProDas possesses a special `Transfer` backbone that all SPI (section 15.6.5) and USART (section 15.7.2) transfer functions as well as many other functions rely on to accomplish their tasks. Under the hood, this backbone is nothing more than a simple interpreter, which interprets transfer related commands, which are encapsulated into one USB packet. This approach increases the efficiency of data exchange between eProDas and periphery (section 14.6.1.2), since many actions like pin toggle, wait for certain pin state, etc. are sent to eProDas as one USB packet instead of several ones.

SPI (sections 15.6.5.1 and 15.6.5.2 as well as sections from 16.5.1 to 16.5.8) and USART (sections 15.7.2 and 16.6.1) transfer functions merely form a USB packet with a proper program and send it to the interpreter, which takes care of the rest. By exploiting the interpreter directly, you can accomplish an efficient data exchange with your periphery in many complicated situations that the users’ set of transfer functions does not and cannot cover due to the unmanageable diversity of chips and communication designs (your own implementations of address buses, decoding logic...).

### 23.2.1 Returning USB packet

Of course, sending commands (and data) is not enough in cases where we would like to receive something from the periphery (e.g. the result of AD conversion). For that matter the interpreter forms an empty USB packet (termed `returning` USB packet) at the beginning of execution of each program to collect returned data into it. When all commands write their data to the packet and the program completed, the data is sent to the host PC in one chunk, which is as efficient as possible. The `returning` packet acts like a FIFO buffer. Each command that has something to send to the host PC adds that data to the end of the current packet contents.

For example, let us assume that your program first receives 3 bytes over SPI bus, then it sends 10 bytes out and finally it receives 5 bytes of data over RS-232 bus. At the end of execution the returning packet is 8 bytes long and the story goes like this. At the beginning of execution the `returning` packet is empty, so the first three bytes (delivered by SPI bus) are written to the very beginning of the packet. The second command merely sends data out of PIC, so nothing is written to the `returning` packet. The third command receives 5 bytes of additional data, which are written to the `returning` packet immediately after the previously written contents.

According to this description, the data is packed into the `returning` packet unlabelled and without any separators. Without knowing the program that executes behind the scenes one cannot tell whether we received 3+5 bytes of data or 8 bytes in one chunk or perhaps 4 times 2 bytes, etc. The bus through which the data is received is also not annotated. It is sole responsibility of your application to properly interpret and demultiplex the contents of the `returning` packet to recover the actual data.

### 23.2.2 Caveat lector

The following subsections describe the exact command codes and the way they are decoded mostly to satisfy your curiosity. It is not impossible that these codes would change in the future versions of eProDas stack and if you form USB packets by hardwiring them directly into your code, you risk that your applications will not be compatible with the future releases of this boring eProDas thing. Instead, rely on accompanying eProDas functions in the section 23.2.5 to build your `Transfer` programs.

When working with `Transfer` backbone, please bear in mind that the interpreter is not a bullet-proof piece of code. It will execute the commands as specified here but it will not detect unimplemented codes, check command parameters or actively seek other errors that your programs may contain. We made it in such way to increase the efficiency of execution.

### 23.2.3 `Transfer` commands

Generally, `Transfer` commands are decoded in more than one stage. The first stage does a partial decoding by examining bits 6 and 7 of the first command byte, according to the following scheme.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | x | x | x | x | x | x | various commands that are further interpreted |
| 0 | 1 | x | x | x | x | x | x | SPI or bidirectional USART transfer |
| 1 | 0 | x | x | x | x | x | x | Toggle bit |
| 1 | 1 | x | x | x | x | x | x | various commands that are further interpreted |

**Figure 106: The first stage of decoding `Transfer` commands.**

When crafting command codes the design goal was to minimize the overall amount of bytes that need to be stuffed into the USB packet to perform certain actions. The most frequently used and time critical commands occupy as little bytes as possible, that is why two important commands `SPI transfer` / `bidirectional USART transfer` and `Toggle bit` (for *ChipSelect* etc.) are recognized in this stage already.

#### 23.2.3.1 SPI transfer (if at least one of the bits 4 and 5 is set)

This command exchanges bytes between eProDas (PIC) and SPI periphery. The meaning of all bits within the command code byte is revealed in the Figure 107.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | o | i | n | n | n | n |

**Figure 107: Encoding of `SPI transfer` command.**

The lower nibble (`nnnn` in the figure) contains (byte_count-1) of data to be transferred to and from periphery. For example, the binary values of 0000, 0001 and 1111 indicate one byte, two bytes and 16 bytes, respectively, of transfer into each direction.

According to SPI protocol data transfer is always bidirectional, which may not always make sense to your application. The meaningless portion of exchanged data can always be discarded but it is a waste of time and a shame to transfer, say, 20 bytes of pointless data over USB cable just to be discarded by your application later on. To remedy the situation there exist two direction bits 5 (`o`: output bit) and 4 (`i`: input bit), through which you specify the meaningful data direction(s).

When bit 5 is set the data to be transferred from eProDas (PIC) to SPI periphery is meaningful. The next (`nnnn`+1) bytes of USB packet that follow the `SPI transfer` command must contain the actual outgoing data. The next `Transfer` command follows these data bytes. However, if bit 5 is cleared the SPI interpreter will not read the outputting data from the USB packet but it will instead transmit zero values; the very next byte of the USB packet must contain the next `Transfer` command.

When bit 4 is set, the data that eProDas (PIC) receives from the periphery is put into the `returning` USB packet (section 23.2.1), otherwise the received data is thrown away.

**Note.** If none of the bits 5 or 4 is set, this command is interpreted as bidirectional USART transfer, which is covered in the section 23.2.3.10.

### 23.2.3.2 Toggle pin

In our opinion toggling of certain pin (like for activating now thousands of times mentioned *ChipSelect*) is one of the most frequent actions that needs to be done for proper progression of data exchange between PIC and periphery. The function `Toggle pin` is here to accomplish this operation. The command is interpreted as follows.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 1 | 0 | **b** | **b** | **b** | **p** | **p** | **p** |

**Figure 108: Encoding of `Toggle pin` command.**

Bits from 0 to 2 (`p`: port) hold port index (valid entries are from 0 for `PORTA` to 4 for `PORTE`), whereas bits from 3 to 5 (`b`: bit) hold the bit or pin of the port to be toggled. The command needs no further arguments and it is completely contained in one byte.

### 23.2.3.3 Repeating the last `SPI transfer` together with `Toggle pin`

Now we are going to reveal you a little secret. Whenever `SPI transfer` command (section 23.2.3.1) is executed, the backbone internally remembers the number of transferred bytes as well as the direction bits (i.e. it remembers the whole byte in the Figure 107). Similarly, when `Toggle pin` command is executed (section 23.2.3.2), the port and bit numbers (the byte in the Figure 108) are remembered. Both remembered values are utilized as a shortcut for further SPI transfers, where toggling of a pin (*ChipSelect*) before and/or after the transfer is done in conjunction. The command code for the task is revealed in the following figure.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | **a** | **b** | **1** | **1** | **0** | **0** |

**Figure 109: Encoding of `Repeat SPI transfer` command.**

First, if bit 4 (`b`: before) is set, the last pin toggle is repeated before `SPI transfer` is initiated.

Second, SPI transfer is executed with the previous `SPI transfer` specifications (number of bytes and directions), however the data is not repeated. The bytes to be transmitted to the periphery must follow (if output was previously done) the command code. Similarly, new bytes that are received from the periphery are written to the returning packet.

Third, if bit 5 (`a`: after) is set, the last pin toggle is repeated after `SPI transfer` is completed.

Such behaviour is handy when you want to perform several SPI transfers to a certain chip together with handling of its *ChipSelect* pin. The first transfer in a row requires three control bytes (two chip select pin toggles and one SPI transfer); however the adjacent transfers only need one control byte for repeating the whole procedure.

By using this feature, the function `ReadExternalAD_MCP320x_All` (section 17.1.1.4) manages to perform AD conversion on all eight AD inputs of integrated circuit MCP3208 by sending only one USB packet to the device. Without it the required number of bytes would exceed the pipe buffer capacity of 44 bytes and two separate USB transfers would be needed.

#### 23.2.3.4 Waiting on a certain pin state

Often you must pause your `Transfer` program until periphery signals through certain PIC's pin that it is ready to proceed (for example, you are waiting for the result of AD conversion before initiating SPI transfer for collecting it). Figure 110 shows the command code to accomplish the described scenario.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | **0** | **0** | **0** | **1** | **s** | **s** |

**Figure 110: Command code for waiting on a certain pin state.**

What we are waiting for is encoded in the bits 0 and 1 according to the Figure 111.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | **0** | **0** | wait for a low state on a pin |
| 0 | 0 | 0 | 0 | 0 | 1 | **0** | **1** | wait for a high state on a pin |
| 0 | 0 | 0 | 0 | 0 | 1 | **1** | **0** | wait for transition from low to high state |
| 0 | 0 | 0 | 0 | 0 | 1 | **1** | **1** | wait for transition from high to low state |

**Figure 111: Encoding the waited for pin state.**

Obviously, the command code reveals the specific pin state/transition but it does not tell anything about the pin in question. For that matter the command must be suffixed by an additional byte which encodes the needed information, according to the following figure.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | **b** | **b** | **b** | **p** | **p** | **p** |

**Figure 112: Encoding of pin on which the state is waited for.**

Bits from 0 to 2 (`p`: port) hold port index (valid entries are from 0 for `PORTA` to 4 for `PORTE`), whereas bits from 3 to 5 (`b`: bit) hold the bit or pin of the port. If bits 6 and 7 are not zero, the whole pattern (both bytes from the Figure 111 and the Figure 112) is decoded as a completely different command.

#### 23.2.3.5 Delay (if delay parameter is not zero)

If your periphery is too slow to follow the data exchange or for whatever reason you can insert a small pause into the execution of `Transfer` program. The bit pattern is as follows.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | **1** | **d** | **d** | **d** | **d** | **d** |

**Figure 113: Encoding of `Delay` command.**

The bits from 0 to 4 specify the amount of delay; however the zero value is interpreted as different command (section 23.2.3.11). The actual delay is calculated according to the following formula.

$$\text{delay}\left[\text{ns}\right] = \left(15 + 3 \cdot \texttt{ddddd}_2\right) \cdot \frac{1000}{12}$$

After this amount of time passes, the interpreter proceeds with execution of the next command in the USB packet.

### 23.2.3.6 Inter-transfer delay

How about if we need a pause between transfer of each byte in a multi-byte `SPI transfer` command? We would have to resort to sending one byte at a time and insert the delay command between each SPI transfer command. Not terribly practical or labour saving. To improve the situation the interpreter also knows about the inter-transfer delay, which works like this.

If you transfer more than one byte at a time (i.e. the field `nnnn` of the `SPI transfer` command is non-zero) then after each transferred byte (except the last one) the interpreter automatically insert a preconfigured amount of delay, which is remembered in an internal variable. This variable is always cleared before interpreter starts to process a new USB packet, so by default the SPI transfer rate is as high as possible.

Due to interpreting overhead there are at least 16 PIC's instructions or 1,333 µs of delay between two consecutive transfers when there is no inter-transfer delay. If this is not enough, you can utilize the `Inter-transfer delay` command to increase the period of inactivity. The command by itself does not make any delay (except the interpreting overhead) but it instead sets the internal delay variable that `(Repeat) SPI transfer` command examines to produce delays. The inter-delay setting is valid until the end of the USB packet or the execution of another `Inter-transfer delay` command. The meaning of command bits is as follows.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | **0** | **1** | **d** | **d** | **d** | **d** |

**Figure 114: Encoding of `Inter-transfer delay` command.**

The field $\text{dddd}_2$ determines the amount of the delay whereas the value of zero means no additional delay other than the previously mentioned overhead. A non-zero value imposes additional delay according to the following formula.

$$\text{inter\_delay}\left[\text{ns}\right]=\left(6+3\cdot\text{dddd}_2\right)\cdot\frac{1000}{12}$$

### 23.2.3.7 Toggle, Set and Clear the first bit correction for SPI transfer

If you are one of those guys who read manuals (even as boring and lengthy as this one is) from cover to cover you surely remember that somewhere deep down in the section 15.6.1 we were talking about correction of the length of the first transferred bit in the byte when SPI clock is controlled by timer TMR2. The correction can be toggled on or off when configuring the module, but this setting can also be altered on the fly during execution of `Transfer` program. Here are not one but three functions for the task. Their roles should be self-evident.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | **0** | **0** | **0** | **0** | **0** | **1** | Toggle correction state |
| 0 | 0 | **0** | **0** | **0** | **0** | **1** | **0** | Switch-on correction |
| 0 | 0 | **0** | **0** | **0** | **0** | **1** | **1** | Switch off correction |

**Figure 115: Manipulating "the first bit correction" settings.**

**Note.** The state is remembered in global eProDas variable and influence of the setting extends beyond the current `Transfer` program.

### 23.2.3.8 Support for implementing address buses and alike

Similarly to the commands in the section 16.2 for implementing address buses our `Transfer` interpreter also enables you to implement various address bus schemes with the following three commands. The functionality should be self evident (if you have read the section 16.2, otherwise you probably do not need these functions).

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | **0** | **0** | **1** | **0** | **0** | **0** | ADD port with bitmask |
| 0 | 0 | **0** | **0** | **1** | **0** | **0** | **1** | ROT port with bitmask |
| 0 | 0 | **0** | **0** | **1** | **0** | **1** | **0** | Write port with bitmask |

**Figure 116: Encoding of address bus oriented commands.**

**ADD port with bitmask**

`ADD port with bitmask` adds a specified constant to the current port's latch value and sieves the result by bitmask to affect only the specified range of bits (technically, the affected bits do not need to be adjacent but in practice this is likely to be the only meaningful choice). It also compares the result with supplied `match value` and resets the result to the supplied `reset value` in case of a match.

The first byte that follows the command code should contain the low byte of the port's latch address (**not** port index), which is $89_{16}$ for `PORTA`, $90_{16}$ for `PORTB`, etc (this is specified in PIC's datasheet).

The second byte contains bitmask with those bits set that are affected by the result. For example, the bitmask of $0001\ 1110_2$ specifies that bits from 1 to 4 of latch register are affected by the operation whereas bits 0, 5, 6 and 7 are preserved.

The third byte contains the actual value to be added to the current latch value (sieved by the mask before and after the adding). The fourth and the fifth bytes are the `match value`, to which the result is compared, and the `reset value` to which the result is rolled over in case of the match, respectively.

**ROT port with bitmask**

`ROT port with bitmask` rotates left a specified subgroup of adjacent bits, whereas the other bits are preserved.

The first byte that follows the command code should contain the low byte of the port's latch address (**not** port index), which is $89_{16}$ for `PORTA`, $90_{16}$ for `PORTB`, etc. The second byte should contain the bitmask of all affected bits. This time the bits **must be** adjacent or the function will not work correctly.

**Write port with bitmask**

Finally, the simplest function within this group. It writes a specified constant into the port's latch register and affects only the bits that are specified by bitmask.

The first byte that follows the command code should contain the low byte of the port's latch address (**not** port index), which is $89_{16}$ for `PORTA`, $90_{16}$ for `PORTB`, etc. The second byte is the bitmask of the affected bits. The third byte is the value to be sieved through the bitmask and written to the port.

### 23.2.3.9 USART OUT and USART IN transfer

The following two commands perform OUT (from PIC to periphery) and IN (from periphery to PIC) USART data transfer, respectively.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | **1** | **1** | **0** | **0** | USART OUT transfer |
| 0 | 0 | 0 | 0 | **1** | **1** | **0** | **1** | USART IN transfer |

**Figure 117: Encoding of USART OUT and IN transfer commands.**

Both of these commands must be followed with a byte that specifies number of bytes to transfer. Actually, only bits from 0 to 4 of this byte are taken into account and bits from 5 to 7 must be set to zero, or the whole command code combination is recognized as completely different command. In addition, this byte must not contain the value of zero.

In the case of USART IN transfer, that's all that constitutes the whole command, whereas the USART OUT command must be followed by the specified number of data bytes.

### 23.2.3.10 Full duplex USART transfer

When doing USART transfer either with command USART OUT or USART IN you actually perform simplex or at most half duplex serial transfer. To implement a full duplex transfer use the following command.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | **n** | **n** | **n** | **n** |

**Figure 118: Encoding of full duplex USART transfer command.**

As it is the case with SPI transfer command (section 23.2.3.1) the lower nibble (nnnn) contains (byte_count-1) of data to be transferred to and from USART periphery: the binary values of 0000, 0001 and 1111 indicate one byte, two bytes and 16 bytes, respectively, of transfer into each direction. The command code must be suffixed with (nnnn+1) bytes of data to be transferred to periphery.

This command works as follows. At the beginning two independent counters are initialized to the value of (nnnn+1). Then receive buffer is checked for new received byte of data; if this is the case, the data is written to the returning packet and the reception counter is decreased. After that, one byte is transferred out of PIC and transmission counter is decreased. The story repeats until both counters are greater than zero. As soon as one counter reaches zero value, only the uncompleted action is executed until the other counter also reaches zero.

Contrary to this command, the issuing of separate USART OUT and USART IN commands does not give the same result since during the transmission eProDas does not check the reception buffer and therefore the data that is received during this time is lost, except one byte that PIC can buffer in hardware.

### 23.2.3.11 USART Clear reception buffer

To make sure that USART reception buffer is empty, use the following command to clear the buffer.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

**Figure 119: Encoding of USART Clear reception buffer command.**

### 23.2.3.12 SwitchFrom_SPI_to_USART and SwitchFrom_USART_to_SPI

Due to the unfortunate division of the pin RC7 among SPI and USART module (section 16.6) the need arises for two commands that enable fast switching between SPI and USART mode of operation. The command codes are as follows.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | **1** | **1** | **1** | **0** | switch from SPI to USART |
| 0 | 0 | 0 | 0 | **1** | **1** | **1** | **1** | switch from USART to SPI |

**Figure 120: Encoding of the two SwitchFrom commands.**

The first of the two disables SPI module, configures pin RC7 as digital input and enables USART module, whereas the second one does the opposite: it disables USART module, configures pin RC7 as digital output and enables SPI module.

**Note.** These two commands work under the presumption that both SPI and USART modules are properly configured. The commands do not check whether this is indeed the case, since the intention is to be able to do fast bus switching. If at least one of the two modules is not configured properly, the behaviour of the device is undetermined.

### 23.2.3.13 Set SPI clock and USART baud rate

In general cases, you want to connect many different SPI enabled chips to a single PIC. Those chips can cope with different SPI clock speeds and without any sort of flexibility you would have to resort to a common denominator among them which means that the slowest periphery would dictate the SPI clock and consequently SPI transfer rate. Similarly, there may be the cases when you communicate with various asynchronous peripherals over USART bus and for that matter it is useful to be able to change the baud rate of the bus on the fly to adopt it to various speeds that peripherals can deal with.

Fortunately, the interpreter can change both settings at your will. The first byte of commands for the tasks is a special prefix byte (Figure 121) that announces a subset of commands to which `Set_SPI_clock` and `Set_USART_BaudRate` belong.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | **0** | **0** | **1** | **0** | **1** | **1** |

**Figure 121: Prefix for `Set SPI clock`, `Set USART Baud Rate` and other commands.**

The next byte that follows the prefix specifies the command and its potential additional parameters according to the following scheme. All don't care bits `x` should be set to zero to maintain future compatibility. As you can see, the high nibble of the command holds the actual command code, which is $0000_2$ for the `Set SPI clock` command and $0001_2$ for `Set USART baud rate` command.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | x | x | **0** | **0** | SPI clock of 12 MHz |
| 0 | 0 | 0 | 0 | x | x | **0** | **1** | SPI clock of 3 MHz |
| 0 | 0 | 0 | 0 | x | x | **1** | **0** | SPI clock of 750 kHz |
| 0 | 0 | 0 | 0 | p | p | **1** | **1** | SPI clock determined by timer TMR2 |
| 0 | 0 | 0 | 1 | x | x | x | **0** | low speed USART baud rate (prescaler of 16) |
| 0 | 0 | 0 | 1 | x | x | x | **1** | high speed USART baud rate (prescaler of 4) |

**Figure 122: Encoding of `Set SPI clock` and `Set USART baud rate` commands.**

The first three entries of the command `Set SPI clock` are straightforward to grasp, whereas the fourth one is explained in section 15.6.1; in this case the `pp` bits determine the prescaler of TMR2, according to the next figure. If and only if the SPI clock is determined by the timer TMR2 an additional byte must follow the command, which specifies TMR2's period register. You do not have to calculate these parameters by hand since the function `Transfer_CMD_SPI_ClockSpeed` (section 23.2.5.18) does it for you.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | **0** | **0** | 1 | 1 | prescaler is 1 |
| 0 | 0 | 0 | 0 | **0** | **1** | 1 | 1 | prescaler is 4 |
| 0 | 0 | 0 | 0 | **1** | **0** | 1 | 1 | prescaler is 16 |
| 0 | 0 | 0 | 0 | **1** | **1** | 1 | 1 | *reserved for further expansions* |

**Figure 123: Encoding of TMR2's prescaler.**

In the case of USART baud rate settings the LSB of command code determines whether low or high speed USART regime is selected. In addition, the command must be suffixed by two bytes that constitute a 16-bit divider of 48 MHz clock. Please, read the section 15.7.1.5 for the explanation. You do not have to calculate these parameters by hand since the function `Transfer_CMD_USART_BaudRate` (section 23.2.5.20) does it for you.

### 23.2.3.14 Configure parameters of SPI clock

Besides selecting an appropriate frequency of SPI clock we often need to change other related parameters, which the following command enables us to do.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | **s** | **i** | 1 | 1 | 1 | 0 | configure parameters of SPI clock |

**Figure 124: Encoding of TMR2's prescaler.**

Bit 4 (i) holds the requested idle state of SPI clock signal, whereas bit 5 (s) determines sampling instant of SPI input signal. When the latter is 0, the signal is sampled in the middle of clock period whereas in the opposite case it is sampled at the end of it. These two settings correspond to the parameters `ClockIdleStateHigh` and `IN_SampleAtTheEnd`, respectively, of the function `SetInternalSPI_MasterMode` (section 15.6.1).

### 23.2.3.15 Read and Write ports

Undoubtedly, reading and writing of digital I/O ports is a frequent task in the majority of microprocessor and microcontroller systems. Transfer backbone supports this activity with the command that enables you to read and write the selected PIC's ports. Command code is as follows.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | WE | WD | **0** | **0** | **0** | **0** |
| WC | WB | WA | RE | RD | RC | RB | RA |

**Figure 125: Encoding of `Read and Write ports` command.**

With bits `Rx` set to 1, you instruct reading of the respective ports, whereas bits `Wx` set to 1 denote writing to the respective ports. So, to read ports B, C and E and write to ports D and E, set bits `RB`, `RC`, `RE`, `WD` and `WE`, but leave other selection bits at zero.

As you can see, with two control bytes you can select arbitrary combination of ports to be read and written to, according to the minimalist philosophy of the backbone. The two-byte command must be followed with as many additional bytes as there are `Wx` bits set. The sequence of writing (and hence fetching the values from USB packet) is from port A to port E. The command adds as many bytes to the returning USB packets as there are `Rx` bits set. Again, the sequence of reading is from port A to port E.

**Note.** Ports A, C and E do not have (available) the last two pins (6 and 7), which this function exploits for delivering the current state of comparators in these two bits (comparator 1 in bit 6 and comparator 2 in bit 7). Therefore, if you read any of the three ports and you are also interested in the state of comparators, you can spare execution of one command.

### 23.2.3.16 Read and XOR ports

This command works in the same way as the previous one does, but it instead performs the XOR operation between port's latch and the specified value. This way you can change the state of certain pins and preserve the rest without first reading the old value.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | XE | XD | **0** | **0** | **0** | **1** |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| XC | XB | XA | RE | RD | RC | RB | RA |

**Figure 126: Encoding of `Read and XOR ports` command.**

**Note.** Ports A, C and E do not have (available) the last two pins (6 and 7), which this function exploits for delivering the current state of comparators in these two bits (comparator 1 in bit 6 and comparator 2 in bit 7). Therefore, if you read any of the three ports and you are also interested in the state of comparators, you can spare execution of one command.

### 23.2.3.17 Write and XOR TRIS registers

When working with digital ports you often need to change the direction of certain pins. The task can be accomplished either by writing new configuration value into TRIS registers or by XOR-ing the current value with a specified parameter. The following command enables you to do both task simultaneously.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | XE | XD | **0** | **0** | **1** | **0** |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| XC | XB | XA | WE | WD | WC | WB | WA |

**Figure 127: Encoding of `Write and XOR TRIS registers` command.**

The command code must be followed with as many bytes as there are bits `Wx` and `Xx` that are set. Writing is done first and XOR-ing after that (in both cases from TRISA to TRISE), which must be respected with the sequence of values that follow the command code.

### 23.2.3.18 Read all internal AD channels

With the following command you can read all internal AD inputs; the `xx` bits must be kept at zero to maintain future compatibility. The PIC automatically examines the current configuration of internal AD module and deduces which channels must be read. Channels are read from AN0 upwards to the maximal configured AD channel.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | **x** | **x** | **0** | **0** | **1** | **1** |

**Figure 128: Encoding of `Read ALL internal AD channels` command.**

For each configured AD channel, two bytes with the result of AD conversion are added to the returning USB packet. Bits from 0 to 7 of the result are contained in the first byte, whereas bits 8 and 9 are contained in the two LSB bits of the next byte.

**Note 1.** To spare the transmitted bytes, two MSB bits of the second byte are **always** written to with the state of internal comparators, so take care to **mask out** these bits when reading AD results. Therefore, if you read AD results you do not need to send command for reading comparators and the returning USB packet needs not to be occupied with the comparators' states separately.

**Note 2.** Regarding previous note, this command does not work properly if left alignment of AD result is selected (section 15.2.2).

### 23.2.3.19 Read selected internal AD channels

Instead of reading all AD channels you may want to read any subset of them according to your preferences. Here comes the command for the task.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | **x** | **x** | **0** | **1** | **0** | **0** |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| AN7 | AN6 | AN5 | AN4 | AN3 | AN2 | AN1 | AN0 |
| | | | | | | | |
| x | x | x | AN12 | AN11 | AN10 | AN9 | AN8 |

**Figure 129: Encoding of `Read selected internal AD channels` command.**

Simply set the `ANx` bits of the respective channels that you want to read and take care that you leave `x` bits cleared for the sake of future compatibility. For each set `ANx` bit, two bytes with the result will be added to the returning USB packet, as it was explained in the previous subsection.

**Note 1.** To spare the transmitted bytes, two MSB bits of the second byte are **always** written to with the state of internal comparators, so take care to mask these bits when reading AD results. Therefore, if you read AD results you do not need to send command for reading comparators and the returning USB packet needs not to be occupied with the comparators' states separately.

**Note 2.** Regarding previous note, this command does not work properly if left alignment of AD result is selected (section 15.2.2).

### 23.2.3.20 Read selected internal AD channel

Use the following command to read one internal AD channel, selected by the `cccc` bit combination. Comparators' state is returned in the two MSB bits of the 16-bit returned value and therefore this command does not work properly if left alignment of AD result is selected (section 15.2.2).

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | **x** | **x** | **0** | **1** | **1** | **0** |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | c | c | c | c | 0 | 0 |

**Figure 130: Encoding of `Read selected AD channel` command.**

### 23.2.3.21 Read selected internal AD channel from AN0 to AN3

Here comes exactly the same command as the previous section described, with the difference that only AD channels from AN0 to AN3 can be accessed, by means of which we spare one byte of command code, as the following figure reveals. The selected channel is encoded in the `cc` bit field.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | c | c | 0 | 1 | 0 | 1 |

**Figure 131: Encoding of `Read selected AD channel from AN0 to AN3` command.**

### 23.2.3.22 Acquire the next AD channel in chain and read it

The `Transfer` backbone also knows how to work with data chains (section **Error! Reference source not found.**). The currently describing function acquires the next channel in chain and reads it (plus comparators).

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | x | x | 0 | 1 | 1 | 1 |

**Figure 132: Encoding of `Acquire the next AD channel in chain and read it` command.**

### 23.2.3.23 Read the currently acquired AD channel and acquire the next one in the chain

This time the sequence of events is reversed in comparison to the previous function: firstly, the currently acquired AD channel is read (plus comparators) and then the next AD channel in chain is acquired. The command code is as follows.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | x | x | 1 | 0 | 0 | 0 |

**Figure 133: Encoding of `Read AD channel and acquire the next one` command.**

### 23.2.3.24 Acquire the first channel in AD chain

The command enables you to reset (not clear) the AD channel sequence and start from the beginning.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | x | x | 1 | 0 | 1 | 1 |

**Figure 134: Encoding of `Acquire the first channel in AD chain` command.**

### 23.2.3.25 Acquire the first channel in AD chain & read it

The same functionality as before but this time we also read the first AD channel (plus comparators) in addition to the acquisition step.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | x | x | 1 | 1 | 0 | 0 |

**Figure 135: Encoding of `Acquire the first channel in AD chain & read it` command.**

### 23.2.3.26 Read internal comparators

To read the current state of the internal comparators, use the following command. The state is returned in two MSB bits of the returned byte. You need to rely on this command only if you do not read AD channel(s) at a suitable time, since all commands for reading AD, also deliver comparator's state.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | x | x | 1 | 0 | 0 | 1 |

**Figure 136: Encoding of `Read internal comparators` command.**

### 23.2.3.27 Configure internal voltage reference

To (re)configure the PIC's internal voltage reference, issue the command in the Figure 137 followed by the byte to be copied to the voltage reference control register. Do now worry, if you haven't read PIC's datasheet and you do not know how the register looks like, since eProDas functions hide technical details from you. Just, keep on reading.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | **x** | **x** | **1** | **0** | **1** | **0** |

**Figure 137: Encoding of `Configure internal voltage reference` command.**

### 23.2.3.28 Exit

The `Exit` command stops the interpreter and sends the `returning` packet data (if any) to the host PC. This command is optional since the interpreter can stop itself when it reaches the end of the USB packet with the program.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | **0** | **0** | **0** | **0** | **0** | **0** |

**Figure 138: Encoding of `Exit` command.**

## 23.2.4 Working with `Transfer` backbone

Now that we know the capabilities of the interpreter and its commands at your exposal, it is time to learn how to send the program to the interpreter and equally important how to write programs without tedious hard coding of the previously presented command codes into the USB packets. Let us start with the function for sending an already finished program to the `Transfer` interpreter.

### 23.2.4.1 TransferBackbone

Prototype in C/C++

```
int eProDas_TransferBackbone(unsigned int DeviceIdx, unsigned char *CommandPacket,
  unsigned char *ReturningPacket, unsigned int CommandPacketLength,
  unsigned int ReturningPacketLength);
```

Prototype in Delphi

```
function eProDas_TransferBackbone(DeviceIdx: LongWord; CommandPacket: PByte;
  ReturningPacket: PByte; CommandPacketLength: LongWord;
  ReturningPacketLength: LongWord):integer;
```

Prototype in VisualBasic

```
eProDas_TransferBackbone
```

Call this function to send your already prepared `Transfer` program to the interpreter for immediate execution. The parameter `CommandPacket` points to a buffer to which you previously stored the program. Also, you need to set the parameter `CommandPacketLength` to the length of your program in bytes (**not** in instructions).

In addition, if your `Transfer` program returns any data, you must allocate/provide another buffer for reception of the returning packet; the pointer `ReturningPacket` holds the starting address of this buffer. The maximal amount is 44 bytes but the actual number depends on your program, of course. The exact number of expected returned bytes must be supplied through the parameter `ReturningPacketLength`. If your program does not return any data, set `ReturningPacketLength` to 0 and `ReturningPacket` to 0 (null pointer).

**Note.** Take absolute care that the value of the parameter `ReturningPacketLength` equals the actual number of returned bytes according to your program or various bad things will happen. If you specify a non-zero value for the parameter and there are no returned bytes, then `eProDas.DLL` will wait for the results forever. If you specify a zero value but there are returned bytes, then eProDas device will try to send the packet that `eProDas.DLL` will not collect and the USB pipe will come out of sync; eProDas will deadlock, crash or behave erratically in the future. Finally, if there are readings but their number

does not match the non-zero value of `ReturningPacketLength`, then you will simply face the error `eProDas_Error_DataCorruptionError`.
One note that is worth remembering is that `Transfer` programs (i.e. `CommandPacket` buffers) remain intact and therefore they can be recycled. Your application can prepare programs in advance and then execute them repeatedly as many times as needed. If you output some data through these packets you only need to refill the changing part of the packet (for example, the bytes that follow the `SPI transfer` command with `o` bit set; see section 23.2.3.1) and leave everything else untouched.

### 23.2.5 Building `Transfer` program packets

As the first step allocate program buffer of capacity 43 bytes; even if your program is shorter, do not waste your precious time counting the number of needed bytes, since PC memory is cheaper than your labour. Even better approach is to allocate a slightly larger buffer (say 64 bytes) so that you can catch potential overflows during program preparation without memory violations or application crashes.

Further, allocate one pointer variable of the appropriate type (pointer to unsigned char in C/C++, pointer to byte in Delphi…) and initialize it to the beginning of the buffer. Now you are ready to utilize the following functions for filling in the packet.

#### 23.2.5.1 Transfer_CMD_SPI_Transfer

Prototype in C/C++

```
int eProDas_Transfer_CMD_SPI_Transfer(unsigned int Send, unsigned int Receive,
  unsigned int Length, unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_SPI_Transfer(Send: LongWord; Receive: LongWord;
  Length: LongWord; var CmdBuffer: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_SPI_Transfer
```

This function fills in the packet with `SPI Transfer` command (section 23.2.3.1). A non-zero value of the parameter `Send` indicates that you want to send true data to the periphery. Similarly, a non-zero value of the parameter `Receive` means that you want to receive the data that periphery sends to the PIC. The `Length` indicates the number of bytes to transfer (from 1 to 16).

The proper `SPI transfer` command code gets synthesized according to your specifications and written to the byte in your buffer that is pointed to by the pointer `CmdBuffer`. Then the pointer gets incremented by means of which it points to the next to be written byte in your buffer. You can always calculate the current length of your SPI program by subtracting this pointer from the beginning of the buffer.

If you intend to reuse this program by executing it many times, you should remember the returned pointer value in another variable to be able to refill the data to be transferred to SPI periphery (only if you specify a non-zero value for `Send`, of course).

Now that the `SPI transfer` command is in place, you can fill in the bytes to be sent (again, only if you perform the outgoing transfer). Start with the address that the pointer `CmdBuffer` points to, and increment it after each written byte. At the end, this pointer must point to the next empty byte in program buffer if you intend to add further commands to the packet.

### 23.2.5.2 Transfer_CMD_SPI_Transfer_AutoFill

Prototype in C/C++

```
int eProDas_Transfer_CMD_SPI_Transfer_AutoFill(unsigned int Receive,
  unsigned int Length, unsigned char *&CmdBuffer, unsigned char *OutData);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_SPI_Transfer_AutoFill (Receive: LongWord;
  Length: LongWord; var CmdBuffer: PByte; OutData: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_SPI_Transfer_AutoFill
```

...a slight variation of the previous function, where not only a `SPI transfer` command code but also your data that is pointed to by `OutData` is being automatically added to the current packet. Obviously, if you call this function you intend to send some data out so the `Send` parameter of previous function is implicitly assumed to be non-zero and needs not to be specified.

### 23.2.5.3 eProDas_Transfer_CMD_TogglePin

Prototype in C/C++

```
int eProDas_Transfer_CMD_TogglePin(unsigned int Port, unsigned int Pin,
  unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_TogglePin(Port: LongWord; Pin: LongWord;
  var CmdBuffer: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_TogglePin
```

The function adds a `Toggle pin` command code (section 23.2.3.2) to the packet. Parameters `Port` and `Pin` specify the pin to be toggled.

### 23.2.5.4 Transfer_CMD_SPI_RepeatTransfer

Prototype in C/C++

```
int eProDas_Transfer_CMD_SPI_RepeatTransfer(unsigned int TogglePinBefore,
  unsigned int TogglePinAfter, unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_SPI_RepeatTransfer(TogglePinBefore: LongWord;
  TogglePinAfter: LongWord; var CmdBuffer: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_SPI_RepeatTransfer
```

This function adds a `Repeat SPI transfer` command code (section 23.2.3.3) to the packet. With parameters `TogglePinBefore` and `TogglePinAfter` you specify, whether you also want to repeat the last pin toggle command before and/or after the SPI transfer.

#### 23.2.5.5 Transfer_CMD_SPI_RepeatTransfer_AutoFill

Prototype in C/C++

```
int eProDas_Transfer_CMD_SPI_RepeatTransfer_AutoFill(unsigned int TogglePinBefore,
  unsigned int TogglePinAfter, unsigned int Length, unsigned char *&CmdBuffer,
  unsigned char *OutData);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_SPI_RepeatTransfer_AutoFill(
  TogglePinBefore: LongWord; TogglePinAfter: LongWord; Length: LongWord;
  var CmdBuffer: PByte; OutData: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_SPI_RepeatTransfer
```

This time not only a `Repeat SPI transfer` command code but also your data that is pointed to by `OutData` is being automatically added to the current packet.

The value of the `Length` parameter must equal the length of the previous SPI transfer.

#### 23.2.5.6 Transfer_CMD_WaitPinState

Prototype in C/C++

```
int eProDas_Transfer_CMD_WaitPinState(unsigned int Port, unsigned int Pin,
  unsigned int State, unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_WaitPinState(Port: LongWord; Pin: LongWord;
  State: LongWord; var CmdBuffer: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_WaitPinState
```

A way to add the `Wait pin state` command (section 23.2.3.4) to the packet. The state/transition to be waited for is encoded in the `State` parameter (from 0 to 3) as the Figure 111 reveals (examine two LSB bits of the command code).

#### 23.2.5.7 Transfer_CMD_Delay

Prototype in C/C++

```
int eProDas_Transfer_CMD_Delay(unsigned int Delay, unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_Delay(Delay: LongWord;
  var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_Delay
```

...and a way to add the `Delay` command code (section 23.2.3.5) to the packet. Needless to say, the delay is specified through the `Delay` parameter (from 1 to 32).

### 23.2.5.8 Transfer_CMD_InterTransferDelay

Prototype in C/C++

```
int eProDas_Transfer_CMD_InterTransferDelay(unsigned int Delay,
  unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_InterTransferDelay(Delay: LongWord;
  var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_InterTransferDelay
```

...and a way to add the `Inter-Transfer Delay` command code (section 23.2.3.6) to the packet. The delay is again specified through the `Delay` parameter (from 0 to 15).

### 23.2.5.9 Transfer_CMD_ConfigFirstBitCorrection

Prototype in C/C++

```
int eProDas_Transfer_CMD_ConfigFirstBitCorrection(unsigned int Action,
  unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_ConfigFirstBitCorrection(Action: LongWord;
  var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_ConfigFirstBitCorrection
```

If the section 23.2.3.7 attained your attention, then this is the function for you. With the `Action` parameter (from 1 to 3), give the function a hint (according to the two LSB bits in the Figure 115) about what do you want to do with the current state of the first bit correction.

### 23.2.5.10 Transfer_CMD_ADD_PortWithBitmask

Prototype in C/C++

```
int eProDas_Transfer_CMD_ADD_PortWithBitmask(unsigned int Port,
  unsigned int BitMask, unsigned int ValueToAdd, unsigned int MatchValue,
  unsigned int ResetValue, unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_ADD_PortWithBitmask(Port: LongWord;
  BitMask: LongWord; ValueToAdd: LongWord; MatchValue: LongWord;
  ResetValue: LongWord; var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_ADD_PortWithBitmask
```

Use the function to add the specified value to a port and specify the bits to be affected by the operation. The operation and parameters are described in section 23.2.3.8.

### 23.2.5.11 Transfer_CMD_ROT_PortWithBitmask

Prototype in C/C++

```
int eProDas_Transfer_CMD_ROT_PortWithBitmask(unsigned int Port,
  unsigned int StartPin, unsigned int EndPin, unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_ROT_PortWithBitmask(Port: LongWord;
  StartPin: LongWord; EndPin: LongWord; var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_ROT_PortWithBitmask
```

Another function from the same group. By reading the section 23.2.3.8, everything should be clear.

### 23.2.5.12 Transfer_CMD_WritePortWithBitmask

Prototype in C/C++

```
int eProDas_Transfer_CMD_WritePortWithBitmask(unsigned int Port,
  unsigned int BitMask, unsigned int ValueToWrite, unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_WritePortWithBitmask(Port: LongWord;
  BitMask: LongWord; ValueToWrite: LongWord; var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_WritePortWithBitmask
```

And the last function from the same group. Again, please read the section 23.2.3.8 for an explanation.

### 23.2.5.13 Transfer_CMD_USART_OUT_Transfer, Transfer_CMD_USART_IN_Transfer

Prototypes in C/C++

```
int eProDas_Transfer_CMD_USART_OUT_Transfer(unsigned int Length,
  unsigned char *&CmdBuffer);

int eProDas_Transfer_CMD_USART_IN_Transfer(unsigned int Length,
  unsigned char *&CmdBuffer);
```

Prototypes in Delphi

```
function eProDas_Transfer_CMD_USART_OUT_Transfer(Length: LongWord;
  var CmdBuffer: PByte):integer;

function eProDas_Transfer_CMD_USART_IN_Transfer(Length: LongWord;
  var CmdBuffer: PByte):integer;
```

Prototypes in VisualBasic

```
eProDas_Transfer_CMD_USART_OUT_Transfer

eProDas_Transfer_CMD_USART_IN_Transfer
```

These two functions fill in the packet with `USART OUT Transfer` and `USART IN Transfer` command (section 23.2.3.9). The `Length` indicates the number of bytes to transfer (from 1 to 31).

In the case of `USART OUT Transfer` command, you must fill in the bytes to be sent. Start with the address that the pointer `CmdBuffer` points to, and increment it after each written byte. At the end, this pointer must point to the next empty byte in program buffer to be ready to add further commands.

### 23.2.5.14 Transfer_CMD_USART_OUT_Transfer_AutoFill

Prototypes in C/C++

```
int eProDas_Transfer_CMD_USART_OUT_Transfer_AutoFill(unsigned int Length,
  unsigned char *&CmdBuffer, unsigned char *OutData);
```

Prototypes in Delphi

```
function eProDas_Transfer_CMD_USART_OUT_Transfer_AutoFill(Length: LongWord;
  var CmdBuffer: PByte; OutData: PByte):integer;
```

Prototypes in VisualBasic

```
eProDas_Transfer_CMD_USART_OUT_Transfer_AutoFill
```

Similarly to the function `Transfer_CMD_SPI_Transfer_AutoFill` (section 23.2.5.2), the currently described command adds the command `USART OUT Transfer` to the buffer and in addition it copies the data to be transferred from buffer `OutData` to the USB command buffer.

### 23.2.5.15 Transfer_CMD_USART_DuplexTransfer (with optional AutoFill)

Prototypes in C/C++

```
int eProDas_Transfer_CMD_USART_DuplexTransfer(unsigned int Length,
  unsigned char *&CmdBuffer);

int eProDas_Transfer_CMD_USART_DuplexTransfer_AutoFill(unsigned int Length,
  unsigned char *&CmdBuffer, unsigned char *OutData);
```

Prototypes in Delphi

```
function eProDas_Transfer_CMD_USART_DuplexTransfer(Length: LongWord;
  var CmdBuffer: PByte):integer;

function eProDas_Transfer_CMD_USART_DuplexTransfer_AutoFill(Length: LongWord;
  var CmdBuffer: PByte; OutData: PByte):integer;
```

Prototypes in VisualBasic

```
eProDas_Transfer_CMD_USART_DuplexTransfer

eProDas_Transfer_CMD_USART_DuplexTransfer_AutoFill
```

The command for doing the full duplex USART transfer (section 23.2.3.10) is added to the packet. The second function also copies the data to be transferred.

### 23.2.5.16 Transfer_CMD_USART_ClearReceptionBuffer

Prototype in C/C++

```
int eProDas_Transfer_CMD_USART_ClearReceptionBuffer(unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_USART_ClearReceptionBuffer(
  var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_USART_ClearReceptionBuffer
```

Insert command for clearing USART reception buffer (23.2.3.11) by calling this function.

### 23.2.5.17 Transfer_CMD_SwitchFrom_*XXX*_to_*YYY*

Prototypes in C/C++

```
int eProDas_Transfer_CMD_SwitchFrom_SPI_To_USART(unsigned char *&CmdBuffer);

int eProDas_Transfer_CMD_SwitchFrom_USART_To_SPI(unsigned char *&CmdBuffer);
```

Prototypes in Delphi

```
function eProDas_Transfer_CMD_SwitchFrom_SPI_To_USART(
  var CmdBuffer: PByte):integer;

function eProDas_Transfer_CMD_SwitchFrom_USART_To_SPI(
  var CmdBuffer: PByte):integer;
```

Prototypes in VisualBasic

```
eProDas_Transfer_CMD_SwitchFrom_SPI_To_USART

eProDas_Transfer_CMD_SwitchFrom_USART_To_SPI
```

…to add codes of the commands `SwitchFrom_SPI_to_USART` and `SwitchFrom_USART_to_SPI` (section 23.2.3.12) to the command packet.

### 23.2.5.18 Transfer_CMD_SPI_ClockSpeed

Prototype in C/C++

```
int eProDas_Transfer_CMD_SPI_ClockSpeed(unsigned int ClockRegime,
  unsigned int Timer2Period, unsigned int Timer2Prescaler,
  unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_SPI_ClockSpeed(ClockRegime: LongWord;
  Timer2Period: LongWord; Timer2Prescaler: LongWord; var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_SPI_ClockSpeed
```

And now for something completely different. To change the clock speed of SPI transfer in your programs (section 23.2.3.13), add the command to the packet with this function. Clock regime (from 0 to 3) is associated with two LSBs in the Figure 122.

In the case of regime 3, you need to properly specify configuration of timer TMR2 through two additional parameters. The parameter `Timer2Period` (from 1 to 256) specifies the period of timer, whereas the parameter `Timer2Prescaler` (with permissive values of 1, 4 and 16) configures the prescaler.

### 23.2.5.19 Transfer_CMD_SPI_ClockConfig

Prototype in C/C++

```
int eProDas_Transfer_CMD_SPI_ClockConfig(unsigned int IN_SampleAtTheEnd,
  unsigned int ClockIdleStateHigh, unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_SPI_ClockConfig(IN_SampleAtTheEnd: LongWord;
  ClockIdleStateHigh: LongWord; var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_SPI_ClockConfig
```

This one adds the commad for configuring SPI clock parameters (section 23.2.3.14) to the packet.

### 23.2.5.20 Transfer_CMD_USART_BaudRate

Prototype in C/C++

```
int eProDas_Transfer_CMD_USART_BaudRate(unsigned int BaudRate,
  unsigned char *&CmdBuffer, unsigned int &ResultedBaudRate);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_USART_BaudRate(BaudRate: LongWord;
  var CmdBuffer: Pbyte; var ResultedBaudRate: LongWord):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_USART_BaudRate
```

Use this function to insert command for changing USART baud rate (section 23.2.3.13) into program packet. The most appropriate mode of operation (whether `HighSpeed` or `LowSpeed`) and associated divider are automatically calculated for you. However, not all baud rates are possible and for that matter the resulted the best possible match is returned through `ResultedBaudRate` parameter for inspection.

### 23.2.5.21 Transfer_CMD_ReadWritePorts (AutoFill)

Prototype in C/C++

```
int eProDas_Transfer_CMD_ReadWritePorts(unsigned int ReadFlags,
  unsigned int WriteFlags, unsigned char *&CmdBuffer);

int eProDas_Transfer_CMD_ReadWritePorts_AutoFill(unsigned int ReadFlags,
  unsigned int WriteFlags, unsigned char *WriteValues, unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_ReadWritePorts(ReadFlags: LongWord;
  WriteFlags: LongWord; var CmdBuffer: Pbyte):integer;

function eProDas_Transfer_CMD_ReadWritePorts_AutoFill(ReadFlags: LongWord;
  WriteFlags: LongWord; WriteValues: Pbyte; var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_ReadWritePorts

eProDas_Transfer_CMD_ReadWritePorts_AutoFill
```

The two functions help you insert command code for reading and writing I/O ports (section 23.2.3.15) into the packet. The ports to be read are specified by the bit field `ReadPorts` (bit 0 for port A, …, bit 4 for port E). Similarly, the ports to be written to are specified by the bit field `WritePorts` (bit roles are the same as before).

The second function also copies as many values from the array `WriteValues` to the USB packet as there are set bits in the bit field `WritePorts`.

**Note.** Ports A, C and E do not have (available) the last two pins (6 and 7), which this function exploits for delivering the current state of comparators in these two bits (comparator 1 in bit 6 and comparator 2 in bit 7). Therefore, if you read any of the three ports and you are also interested in the state of comparators, you can spare execution of one command.

### 23.2.5.22 Transfer_CMD_Read_XOR_Ports (AutoFill)

Prototype in C/C++

```
int eProDas_Transfer_CMD_Read_XOR_Ports(unsigned int ReadFlags,
  unsigned int XORFlags, unsigned char *&CmdBuffer);
```

```
int eProDas_Transfer_CMD_Read_XOR_Ports_AutoFill(unsigned int ReadFlags,
  unsigned int XORFlags, unsigned char *XORValues, unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_Read_XOR_Ports(ReadFlags: LongWord;
  XORFlags: LongWord; var CmdBuffer: Pbyte):integer;
```

```
function eProDas_Transfer_CMD_Read_XOR_Ports_AutoFill(ReadFlags: LongWord;
  XORFlags: LongWord; XORValues: Pbyte; var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_Read_XOR_Ports
```

```
eProDas_Transfer_CMD_Read_XOR_Ports_AutoFill
```

Precisely the same functionality as in the previous section, except that this time we are dealing with reading and XOR-ing I/O ports (section 23.2.3.16).

**Note.** Ports A, C and E do not have (available) the last two pins (6 and 7), which this function exploits for delivering the current state of comparators in these two bits (comparator 1 in bit 6 and comparator 2 in bit 7). Therefore, if you read any of the three ports and you are also interested in the state of comparators, you can spare execution of one command.

### 23.2.5.23 Transfer_CMD_Write_XOR_TRISes (AutoFill)

Prototype in C/C++

```
int eProDas_Transfer_CMD_Write_XOR_TRISes(unsigned int WriteFlags,
  unsigned int XORFlags, unsigned char *&CmdBuffer);
```

```
int eProDas_Transfer_CMD_Write_XOR_TRISes_AutoFill(unsigned int WriteFlags,
  unsigned int XORFlags, unsigned char *WriteXORValues, unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_Write_XOR_TRISes(WriteFlags: LongWord;
  XORFlags: LongWord; var CmdBuffer: Pbyte):integer;
```

```
function eProDas_Transfer_CMD_Write_XOR_TRISes_AutoFill(WriteFlags: LongWord;
  XORFlags: LongWord; WriteXORValues: Pbyte; var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_Write_XOR_TRISes
```

```
eProDas_Transfer_CMD_Write_XOR_TRISes_AutoFill
```

Precisely the same functionality as in the previous two sections, except that this time we are dealing with writing and XOR-ing TRIS registers (section 23.2.3.17).

### 23.2.5.24 Transfer_CMD_ReadAllInternalADChannels

Prototype in C/C++

```
int eProDas_Transfer_CMD_ReadAllInternalADChannels(unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_ReadAllInternalADChannels(
  var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_ReadAllInternalADChannels
```

Insertion of command code for reading all configured internal AD channels (section 23.2.3.18).

### 23.2.5.25 Transfer_CMD_ReadSelectedInternalADChannels

Prototype in C/C++

```
int eProDas_Transfer_CMD_ReadSelectedInternalADChannels(unsigned int ChannelFlags,
  unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_ReadSelectedInternalADChannels(
  ChannelFlags: LongWord; var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_ReadSelectedInternalADChannels
```

Insertion of command code for reading a selected set of internal AD channels (section 23.2.3.19). The requested channels are indicated by the bit field `ChannelFlags` (bit 0 for AN0, …, bit 12 for AN12).

### 23.2.5.26 Transfer_CMD_ReadInternalADChannel

Prototype in C/C++

```
int eProDas_Transfer_CMD_ReadInternalADChannel(unsigned int Channel,
  unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_ReadInternalADChannel(Channel: LongWord;
  var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_ReadInternalADChannel
```

Insertion of command code for reading one selected internal AD channel (section 23.2.3.20), which is specified by the parameter `Channel`.

### 23.2.5.27 Transfer_CMD_ReadInternalADChannel _0_3

Prototype in C/C++

```
int eProDas_Transfer_CMD_ReadInternalADChannel_0_3(unsigned int Channel,
  unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_ReadInternalADChannel_0_3 (Channel: LongWord;
  var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_ReadInternalADChannel_0_3
```

Precisely the same functionality as with the previous function, except that this time the `Channel` can only be within the range from 0 to 3, by means of which one control byte is spared (section 23.2.3.21).

### 23.2.5.28 Transfer_CMD_AcquireFirstInternalADChannelInChain

Prototype in C/C++

```
int eProDas_Transfer_CMD_AcquireFirstInternalADChannelInChain(
  unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_AcquireFirstInternalADChannelInChain(
  var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_AcquireFirstInternalADChannelInChain
```

Insertion of command code for resetting the channel chain and acquiring the first AD channel in it (section 23.2.3.24).

### 23.2.5.29 Transfer_CMD_AcquireReadFirstInternalADChannelInChain

Prototype in C/C++

```
int eProDas_Transfer_CMD_AcquireReadFirstInternalADChannelInChain(
  unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_AcquireReadFirstInternalADChannelInChain(
  var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_AcquireReadFirstInternalADChannelInChain
```

Insertion of command code for resetting the channel chain and acquiring as well as reading the first AD channel in it (section 23.2.3.25).

### 23.2.5.30 Transfer_CMD_AcquireReadNextInternalADChannelInChain

Prototype in C/C++

```
int eProDas_Transfer_CMD_AcquireReadNextInternalADChannelInChain(
  unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_AcquireReadNextInternalADChannelInChain(
  var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_AcquireReadNextInternalADChannelInChain
```

Insertion of command code for acquiring and reading the next AD channel in channel chain (section 23.2.3.22).

### 23.2.5.31 Transfer_CMD_ReadAcquireNextInternalADChannelInChain

Prototype in C/C++

```
int eProDas_Transfer_CMD_ReadAcquireNextInternalADChannelInChain(
  unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_ReadAcquireNextInternalADChannelInChain(
  var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_ReadAcquireNextInternalADChannelInChain
```

And insertion of command code for reversed operation in comparison to the previous section: reading the currently acquired channel and acquiring the next one in channel chain (section 23.2.3.23).

### 23.2.5.32 Transfer_CMD_ReadInternalComparators

Prototype in C/C++

```
int eProDas_Transfer_CMD_ReadInternalComparators(unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_ReadInternalComparators(
  var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_ReadInternalComparators
```

Insertion of command code for reading the current state of internal voltage comparators (section 23.2.3.26).

### 23.2.5.33 Transfer_CMD_ConfigureInternalVoltageReference

Prototype in C/C++

```
int eProDas_Transfer_CMD_ConfigureInternalVoltageReference(unsigned int Enabled,
   unsigned int Value, unsigned int Range, unsigned int SourceOnPin,
   unsigned int OutputOnPin, unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_ConfigureInternalVoltageReference(Enabled: LongWord;
  Value: LongWord; Range: LongWord; SourceOnPin: LongWord; OutputOnPin: LongWord;
  var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_ConfigureInternalVoltageReference
```

Insertion of command code for configuring the internal voltage reference (section 23.2.3.27). The meaning of parameters is the same as with the function `ConfigureInternalVoltageReference` (section 15.4.1).

### 23.2.5.34 Transfer_CMD_Exit

Prototype in C/C++

```
int eProDas_Transfer_CMD_Exit(unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_Transfer_CMD_Exit(var CmdBuffer: Pbyte):integer;
```

Prototype in VisualBasic

```
eProDas_Transfer_CMD_Exit
```

Generally, you do not need this command since `Transfer` backbone stops itself automatically at the end of the USB packet program. But sometimes you may use it to alter the packet if under certain conditions you want to execute only a portion of packet without building the program afresh.

### 23.2.6 Example `Transfer` program

To better grasp the idea of how to use the `Transfer` backbone, a simplified implementation of the function `Master_SPI_TransferArrayCS` (section 16.5.5) is included as a real world example program.

```
eProDasDLL_API int eProDas_MasterSPI_TransferArrayCS(unsigned int DeviceIdx,
  unsigned char *WriteArray, unsigned char *ReadArray, unsigned int Length,
  unsigned int ChipSelectPort, unsigned int ChipSelectBit) {


  unsigned char SPI_Program[64]; //buffer with soon to be formed SPI program
  unsigned char *pSPI = SPI_Program; //pointer that is needed during program formation

  int Status;

  //the first SPI command: toggle ChipSelect pin to activate the chip
  Status = eProDas_Transfer_CMD_TogglePin(ChipSelectPort, ChipSelectBit, pSPI);
  if(Status!=eProDas_Error_SUCCESS) { return Status; }

  //the second SPI command: do the transfer
  Status = eProDas_Transfer_CMD_Transfer_AutoFill(1, Length, pSPI, WriteArray);
  if(Status!=eProDas_Error_SUCCESS) { return Status; }

  //the third SPI command: toggle ChipSelect pin to deactivate the chip
  Status = eProDas_Transfer_CMD_TogglePin(ChipSelectPort, ChipSelectBit, pSPI);
  if(Status!=eProDas_Error_SUCCESS) { return Status; }

  //until now the hardware has not been touched yet, only the SPI program
  //has been written to the SPI_Program buffer

  //the following command now sends the program to the SPI backbone,
  //which does the true SPI transfer & ChipSelect toggling.
  Status = eProDas_TransferBackbone(DeviceIdx, SPI_Program,
    ReadArray, unsigned int(pSPI-SPI_Program), Length);

  return Status;
}
```

As you can see, the operation is rather straightforward. First, we build a desired `Transfer` program by adding the commands to the program buffer `SPI_program`. The first command toggles a *ChipSelect* pin to activate the chip, the second command instructs the actual transfer and the third command returns *ChipSelect* to the inactive state. So far, only the buffer `SPI_program` has been affected and no action has been taken on a hardware level. If the function aborts at this stage neither eProDas device nor the backbone would ever know that someone was trying to form a `Transfer` program.

Then, by calling the function `TransferBackbone` the program becomes alive and the hardware is affected by execution of the specified sequence of steps by the backbone. At the end the status of the execution is checked for reporting it to the user or in elaborate cases to establish the rescue plan.

## 23.3 WaitState backbone

This backbone enables you to wait for arbitrary combination of conditions on digital I/O ports and pins as well as the state of voltage comparators separately or in a combination. Similarly to the previously described `Transfer` backbone, this one is implemented as a dedicated interpreter that sequentially executes your commands to achieve the goal of yours.

Generally, when merely waiting for a prescribed state, a one-way communication between your application and eProDas device suffices, since you merely need to instruct the device about the waiting conditions. However, there are a few `WaitState` functions that do not wait for a prescribed state but they instead wait that the state becomes stable (i.e. the state does not change for a while). In such cases you are probably interested in the particular state that exerts stability. For that matter, the `WaitState` interpreter forms a returning USB packet during execution of your program, where all such states are collected and returned to your application. Similar to the `Transfer` backbone, this packet is termed `returning` packet and it behaves in the same way as the section 23.2.1 describes.

Please, read the warnings in section 23.2.2 about `Transfer` backbone, which are relevant to the `WaitState` backbone as well.

### 23.3.1 WaitState commands

Similar to a well-designed pipelined processor (we are kidding; of course), `WaitState` commands are interpreted in more than one stage. The first stage does a partial decoding by examining only bit 7 of the first command byte, according to the following scheme.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|
| 1 | s | b | b | b | p | p | p | wait for pin state |
| 0 | c | c | c | a | a | a | a | various commands that are further interpreted |

**Figure 139: The first stage of decoding `WaitState` commands.**

Again, when crafting `WaitState` command codes the design goal was to minimize the overall amount of bytes that need to be stuffed into the USB packet to perform desired actions. Arguably, the most frequently used command within the scope of this backbone is waiting for a certain state on a pin. Fortunately, this whole command together with its parameters can be squeezed into one byte and we did not hesitate to take the advantage of this fact, as the top row of the Figure 139 reveals.

#### 23.3.1.1 Wait for pin state

This command monitors a user specified pin in a tight loop until a prescribed state occurs. If the state does not occur the eProDas device waits for it indefinitely. The position of bits or bit fields in the command byte is specified in the top row of the Figure 139. The bit `s` specifies the waited for state of a pin, whereas bit fields `ppp` (port) and `bbb` (bit) specify port and pin index, respectively, of a selected pin. If you jumped directly to this section and you are unfamiliar with pin specification, please examine the Figure 112 (page 291) and its accompanying explanation.

#### 23.3.1.2 Second stage of command interpretation

If the most significant bit of the command byte is 0 (the second row of the Figure 139), the command is further interpreted according to the `ccc` (command) bit field, as the following figure reveals. The bit field `aaaa` (argument) holds potential additional arguments of the command.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|
| 0 | **0** | **0** | **0** | **c** | **c** | **c** | **c** | various commands that are further interpreted |
| 0 | **0** | **0** | **1** | **s** | **s** | **m** | **m** | wait for comparators state |
| 0 | **0** | **1** | **0** | **x** | **p** | **p** | **p** | wait for port state |
| 0 | **0** | **1** | **1** | **s** | **s** | **m** | **m** | repetitively wait for comparators state |
| 0 | **1** | **0** | **0** | **x** | **p** | **p** | **p** | repetitively wait for port state |
| 0 | **1** | **0** | **1** | **x** | **p** | **p** | **p** | wait for stabilized port state |
| 0 | **1** | **1** | **0** | **x** | **x** | **m** | **m** | wait for stabilized comparators state |
| 0 | **1** | **1** | **1** | **x** | **x** | **x** | **x** | *reserved for further expansions* |

**Figure 140: Encoding of commands with MSB equal to zero.**

### 23.3.1.3 Wait for port state

Instead of waiting for a certain state on an isolated pin, you can wait for an arbitrary state on a whole digital I/O port. The form of the command for the task is revealed in the third row of the Figure 140. The bit field `ppp` (port) holds port index whereas bit 3 is currently ignored but it should be set to 0 to maintain compatibility with future versions of the interpreter. The command in its complete incarnation is specified in the following figure.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | **p** | **p** | **p** | wait for port state |
| **m** | **m** | **m** | **m** | **m** | **m** | **m** | **m** | bitmask |
| **s** | **s** | **s** | **s** | **s** | **s** | **s** | **s** | sought for state |

**Figure 141: The entire form of the command `Wait For Port State`.**

Generally, we are not interested in all pins of the port and for that matter the second byte of the command specifies bitmask of the bits that need to be taken under examination (value of 1). The third byte holds the state that we are waiting for. If the state does not occur, the device waits forever.

### 23.3.1.4 Wait for comparators' state

Digital states are not the only thing that we may want to wait for. When we are talking about data acquisition systems, there is an emphasis on analog world and for that matter we should be able to wait on certain analog conditions, too. For the start, let us see how to wait on a prescribed state of internal PIC's comparators. The form of the command is revealed in the second row of the Figure 140.

Since there are two comparators that are built into the PIC, we need two bits `ss` (state) to specify their desired state. Further, with bits `mm` (mask), you can specify which one or both of them interest you (the value of 1). Comparator 1 has always lower bit number in all `ss` and `mm` schemes. If both `mm` bits are 0, the command is completely useless and for that matter this particular combination is reserved for future expansions.

**Note.** If the comparators' module is not enabled or it is in mode 0 (please, see the Figure 37 on page 84) prior to executing a `WaitState` program, this command does not wait on anything.

### 23.3.1.5 Wait for internal AD converter result

To delve further into analog world we certainly need the possibility of waiting for some AD result to enter a prescribed interval of values. The following command exists for this particular task.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | **0** | **0** | **0** | **1** | wait for AD conversion result |
| h | l | c | c | c | c | x | x | channel and thresholds specification |

**Figure 142: The form of the command `Wait For AD Result`.**

The bit field `cccc` (channel) specifies the channel of internal AD converter to monitor for the prescribed state, whereas the value of 1 of bits `h` and `l` selects the operational `high` and/or `low` threshold, respectively. At least one of the thresholds must be active or the behaviour of this command is unpredictable (and the combination reserved for future expansions). When one of the bits `l` or `h` is set, the command must be followed by two little-endian bytes that hold 10-bit threshold value. If both `l` and `h` bits are set, the command must be followed by four additional bytes, where the first two specify the `low` threshold and the remaining tow the `high` one.

If bit `l` is set, the command waits until the AD result becomes larger than the specified `low` threshold. If bit `h` is set, the command waits until the AD result becomes smaller than the prescribed `high` threshold. If both bits `l` and `h` are set, both of the just stated conditions must be fulfilled before the waiting is over. The AD result that fulfils the stated conditions is returned to the user's application through the returning packet.

**Note 1.** If the internal AD converter module is not enabled and configured prior to executing a `WaitState` program, this command does not wait for anything and returns gibberish instead of meaningful AD result.

**Note 2.** This command does not work properly if left alignment of AD result is selected (section 15.2.2).

### 23.3.1.6 Repetitively wait for port state

Sometimes you do not only want to wait for a certain port state but you would also like to make sure that the waited for state is stable, i.e. that it is not a result of a noise or a spike on a line. This functionality can be achieved by reading the state repetitively several times and checking that all readings are equal. If only one examined value differs from the others, the whole process is repeated from the beginning. The command code for the task is revealed in the following figure.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | **p** | **p** | **p** | repetitively wait for port state |
| **m** | **m** | **m** | **m** | **m** | **m** | **m** | **m** | bitmask |
| **s** | **s** | **s** | **s** | **s** | **s** | **s** | **s** | sought for state |
| **low byte** | | | | | | | | two's complement of repetitions (LSB) |
| **high byte** | | | | | | | | two's complement of repetitions (MSB) |

**Figure 143: The form of the command `Repetitively Wait For Port State`.**

The second and the third bytes are semantically equal to the respective bytes of the command `WaitForPortState`. The last two bytes specify the two's complement of the number of adjacent readings that must not differ in order to recognize the state as stable.

There is a delay of about 6 PIC instructions (500 ns) between two adjacent readings of the state, so by specifying the maximal possible number of repetitions (65,535) it is possible to demand that the state is unchanged for more than 30 ms before it is declared stable. If only one reading during this time differs, the whole waiting process repeats from the beginning.

### 23.3.1.7 Repetitively wait for comparators' state

Similarly to the previous command, we can make sure that the state of comparators is unchanged for a while by repetitively examining it, before we declare it stable. The command for the task works in the same way as the plain monitoring of comparators (section 23.3.1.4) does, except that it adds repetitions to the whole process as described in the previous section. The command code for the task follows.

| 7 6 5 4 3 2 1 0 | |
|---|---|
| 0 0 1 1 s s m m | repetitively wait for comparators state |
| low byte | two's complement of repetitions (LSB) |
| high byte | two's complement of repetitions (MSB) |

**Figure 144: The form of the command** `Repetitively Wait For Comparators State.`

### 23.3.1.8 Wait for stabilized port state

Here comes the third possibility of waiting for port state. Again, the process is repetitive, however this time we only wait for the state to stabilize and we do not prescribe any particular value that we are waiting for. The only thing that matters is that all prescribed number of adjacent port's readings return equal value. The command code for the task is as follows.

| 7 6 5 4 3 2 1 0 | |
|---|---|
| 0 1 0 1 x p p p | wait for stabilized port state |
| m m m m m m m m | bitmask |
| low byte | two's complement of repetitions (LSB) |
| high byte | two's complement of repetitions (MSB) |

**Figure 145: The form of the command** `Wait For Stabilized Port State.`

The waited for state is not prescribed in advance. This means that your application cannot know the port's state upon completion of this command. Reading the port afterwards does not provide a solution, since the state might change a fraction of microsecond after being recognized as stable. Nonetheless, it may be important for your application to learn about the stable state and for that matter this information is added to the `returning` USB packet (explained in the section 23.3), which your application may examine.

### 23.3.1.9 Wait for stabilized comparators' state

Here we have a function that implements similar functionality for comparators.

| 7 6 5 4 3 2 1 0 | |
|---|---|
| 0 1 1 0 x x m m | repetitively wait for comparators state |
| low byte | two's complement of repetitions (LSB) |
| high byte | two's complement of repetitions (MSB) |

**Figure 146: The form of the command** `Wait For Stabilized Comparators State.`

**Note.** The reached stable comparators' state is written to the `returning` USB packet (explained in the section 23.3) and is therefore available to your application.

#### 23.3.1.10 Exit

The `Exit` command stops the interpreter and sends the `returning` packet data (if any) to the host PC. This command is optional since the interpreter can stop itself when it reaches the end of the USB packet with the program.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | **0** | **0** | **0** | **0** |

**Figure 147: Encoding of `Exit` command.**

### 23.3.2 Working with `WaitState` interpreter

Now that we know about the commands that are available, it is time to learn how to send the `WaitState` program to the interpreter and how to write `WaitState` programs without tedious hard coding of the previously presented command codes into the USB packets. This material is basically copy-pasted & adjusted from the analogous section 23.2.4 about `Transfer` backbone; if you have studied that backbone already, this section will certainly be even more boring to read than the rest of this tedious book. Let us start with the function for sending a ready-to-go program to the `WaitState` interpreter.

#### 23.3.2.1 WaitStateBackbone

Prototype in C/C++

```
int eProDas_WaitStateBackbone(unsigned int DeviceIdx, unsigned char *CommandPacket,
  unsigned char *ReturningPacket, unsigned int CommandPacketLength,
  unsigned int ReturningPacketLength);
```

Prototype in Delphi

```
function eProDas_WaitStateBackbone(DeviceIdx: LongWord;
  CommandPacket: PByte; ReturningPacket: PByte; CommandPacketLength: LongWord;
  ReturningPacketLength: LongWord):integer;
```

Prototype in VisualBasic

```
eProDas_MasterSPI_WaitStateBackbone
```

Use this function to send your already prepared `WaitState` program to the interpreter for immediate execution. The parameter `CommandPacket` points to a buffer to which you previously stored the `WaitState` program. Also, you need to set the parameter `CommandPacketLength` to the length of your program (in bytes, **not** in instructions).

In addition, if your `WaitState` program returns any data, you must provide another buffer for reception of the returning packet; the pointer `ReturningPacket` holds the starting address of this buffer. The maximal amount is 44 bytes but the actual number depends on your program, of course. The exact number of expected returned bytes must be supplied through the parameter `ReturningPacketLength`. If your program does not return any data, set `ReturningPacketLength` to 0 and `ReturningPacket` to 0 (null pointer).

**Note.** Take absolute care that the value of the parameter `ReturningPacketLength` equals the actual number of returned bytes according to your program or various bad things will happen. If you specify a non-zero value for the parameter and there are no returned bytes, then `eProDas.DLL` will wait for the results forever. If you specify a zero value but there are returned bytes, then eProDas device will try to send the packet that `eProDas.DLL` will not collect and the USB pipe will come out of sync; eProDas will deadlock, crash or behave erratically in the future. Finally, if there are readings but their number does not match the non-zero value of `ReturningPacketLength`, then you will simply face the error `eProDas_Error_DataCorruptionError`.

One note that is worth remembering is that `WaitState` programs (i.e. `CommandPacket` buffers) remain intact and therefore they can be recycled. Your application can prepare programs in advance and then execute them repeatedly as many times as needed. If you change conditions (like AD thresholds) you only need to refill the changing part of the packet and leave everything else untouched.

### 23.3.3 Building `WaitState` program packets

As the first step allocate `WaitState` program buffer of capacity 43 bytes; even if your program is shorter, do not waste your precious time counting the number of needed bytes, since PC memory is cheaper than your labour. Even better approach is to allocate a slightly larger space (say 64 bytes) so that you can catch potential overflows without triggering memory violations or application crashes.

In addition, allocate one pointer variable of the appropriate type (pointer to unsigned char in C/C++, pointer to byte in Delphi…) and initialize it to the beginning of the buffer. Now you are ready to utilize the following functions for filling in the packet.

#### 23.3.3.1 WaitState_CMD_PinState

Prototype in C/C++

```
int eProDas_WaitState_CMD_PinState(unsigned int Port, unsigned int Pin,
  unsigned int State, unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_WaitState_CMD_PinState(Port: LongWord; Pin: LongWord;
  State: LongWord; var CmdBuffer: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_ WaitState_CMD_PinState
```

This function fills in the packet with `Wait For Pin State` command (section 23.3.1.1). `Port` (0 for PortA, 1 for PortB …) and `Pin` (from 0 to 7) together specify the pin in question. A non-zero value of the parameter `State` indicates that you want to wait for a high pin state (logic 1).

The proper `Wait For Pin State` command code gets synthesized according to your specifications and written to the byte in your buffer that is pointed to by the pointer `CmdBuffer`. Then the pointer gets incremented by means of which it points to the next to be written byte in your buffer. You can always calculate the current length of your `WaitState` program by subtracting this pointer from the beginning of the buffer.

#### 23.3.3.2 WaitState_CMD_PortState

Prototype in C/C++

```
int eProDas_WaitState_CMD_PortState(unsigned int Port, unsigned int BitMask,
  unsigned int State, unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_WaitState_CMD_PortState(Port: LongWord; BitMask: LongWord;
  State: LongWord; var CmdBuffer: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_ WaitState_CMD_PortState
```

This function fills in the packet with `Wait For Port State` command (section 23.3.1.3). `Port` (0 for PortA, 1 for PortB …) specifies the port in question, whereas `BitMask` (form 0 to 255) specifies pins that you are interested in monitoring (a non zero bit value in `BitMask`). The state to be waited for is specified by the `State` parameter, which is automatically sieved by `BitMask`.

#### 23.3.3.3 WaitState_CMD_ComparatorsState

Prototype in C/C++

```
int eProDas_WaitState_CMD_ComparatorsState(unsigned int MonitorComp1,
  unsigned int MonitorComp2, unsigned int State1, unsigned int State2,
  unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_WaitState_CMD_ComparatorsState(MonitorComp1: LongWord;
  MonitorComp2: LongWord; State1: LongWord; State2: LongWord;
  var CmdBuffer: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_ WaitState_CMD_ComparatorsState
```

This function fills in the packet with `Wait For Comparators State` command (section 23.3.1.4). A non-zero value of `MonitorCompX` specifies that the state of the respective comparator `X` should be taken into account; otherwise the comparator is ignored. Note that at least one of the two comparators must not be ignored or the function returns with the error `eProDas_Error_InvalidComparatorMode`.

A non-zero value of `StateX` indicates that we are waiting for high logic output of the respective comparator `X` (only if the associated `MonitorCompX` has a non-zero value, otherwise the parameter is ignored).

#### 23.3.3.4 WaitState_CMD_InternalAD

Prototype in C/C++

```
int eProDas_WaitState_CMD_InternalAD(unsigned int Channel,
  unsigned int LowThreshold, unsigned int HighThreshold,
  unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_WaitState_CMD_InternalAD(Channel: LongWord;
  LowThreshold: LongWord; HighThreshold: LongWord; var CmdBuffer: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_ WaitState_CMD_InternalAD
```

This function fills in the packet with `Wait For Internal AD Converter Result` command (section 23.3.1.5). Parameter `Channel` specifies the internal AD channel (from 0 to 12) to be monitored. The function waits until the AD conversion result on the selected channel is greater than or equal to the `LowThreshold` and simultaneously lower than or equal to the `HighThreshold`. Both thresholds must have a value between 1 and 1023; a value of zero disables the respective threshold. If `HighThreshold` is smaller than `LowThreshold`, the function returns erroneously.

### 23.3.3.5 WaitState_CMD_RepetitivePinState

Prototype in C/C++

```
int eProDas_WaitState_CMD_RepetitivePinState(unsigned int Repetitions,
  unsigned int Port, unsigned int Pin, unsigned int State,
  unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_WaitState_CMD_RepetitivePinState(Repetitions: LongWord;
  Port: LongWord; Pin: LongWord; State: LongWord; var CmdBuffer: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_WaitState_CMD_RepetitivePinState
```

This function fills in the packet with command `Repetitively Wait For Pin State`, which was not described yet, since internally this command is implemented as `Repetitively Wait For Port State` (section 23.3.1.6) with a suitable `BitMask`.

Parameter `Repetitions` (from 1 to 65536) specifies the desired number of adjacent pin readings that return the desired value before the state is declared stable. Other parameters have the same meaning as with the command `WaitState_CMD_PinState` (section 23.3.1.1).

### 23.3.3.6 WaitState_CMD_RepetitivePortState

Prototype in C/C++

```
int eProDas_WaitState_CMD_RepetitivePortState(unsigned int Repetitions,
  unsigned int Port, unsigned int BitMask, unsigned int State,
  unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_WaitState_CMD_RepetitivePortState(Repetitions: LongWord;
  Port: LongWord; BitMask: LongWord; State: LongWord;
  var CmdBuffer: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_WaitState_CMD_RepetitivePortState
```

This function fills in the packet with command `Repetitively Wait For Port State` (section 23.3.1.6). Parameter `Repetitions` (from 1 to 65536) specifies the desired number of adjacent port readings that return the desired value before the state is declared stable. Other parameters have the same meaning as with the command `WaitState_CMD_PortState` (section 23.3.3.2).

### 23.3.3.7 WaitState_CMD_RepetitiveComparatorsState

Prototype in C/C++

```
int eProDas_WaitState_CMD_RepetitiveComparatorsState(unsigned int Repetitions,
  unsigned int MonitorComp1, unsigned int MonitorComp2, unsigned int State1,
  unsigned int State2, unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_WaitState_CMD_RepetitiveComparatorsState(Repetitions: LongWord;
  MonitorComp1: LongWord; MonitorComp2: LongWord; State1: LongWord;
  State2: LongWord; var CmdBuffer: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_WaitState_CMD_RepetitiveComparatorsState
```

This function fills in the packet with command `Repetitively Wait For Comparators State` (section 23.3.1.7). Parameter `Repetitions` (from 1 to 65536) specifies the desired number of adjacent comparator's state readings that return the desired value before the state is declared stable. Other parameters have the same meaning as with the command `WaitState_CMD_ComparatorsState` (section 23.3.3.3).

### 23.3.3.8 WaitState_CMD_StablePortState

Prototype in C/C++

```
int eProDas_WaitState_CMD_StablePortState(unsigned int Repetitions,
  unsigned int Port, unsigned int BitMask, unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_WaitState_CMD_StablePortState(Repetitions: LongWord;
  Port: LongWord; BitMask: LongWord; var CmdBuffer: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_WaitState_CMD_StablePortState
```

This function fills in the packet with command `Wait For Stabilized Port State` (section 23.3.1.8), which works in the same way as the command `WaitState_CMD_RepetitivePortState` (section 23.3.3.6) does, except that this time the desired state is not specified.

### 23.3.3.9 WaitState_CMD_StableComparatorsState

Prototype in C/C++

```
int eProDas_WaitState_CMD_StableComparatorsState(unsigned int Repetitions,
  unsigned int MonitorComp1, unsigned int MonitorComp2, unsigned char *&CmdBuffer);
```

Prototype in Delphi

```
function eProDas_WaitState_CMD_StableComparatorsState(Repetitions: LongWord;
  MonitorComp1: LongWord; MonitorComp2: LongWord; var CmdBuffer: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_WaitState_CMD_StableComparatorsState
```

This function fills in the packet with command `Wait For Stabilized Comparators State` (section 23.3.1.9), which works the same as the command `WaitState_CMD_RepetitiveComparatorsState` (section 23.3.3.7) does, except that this time the desired state is not specified.

### 23.3.4 Example `WaitState` program

To better grasp the idea about how to use the WaitState backbone, we include a simplified implementation of the function `WaitInternalADTransition` (section 16.3.2) as a real world example program. As with `Transfer` example program (section 23.2.6), we can conclude that in the first step the program is formed and then it is chewed by the backbone to achieve the stated result.

```
int eProDas_WaitInternalADTransition(unsigned int DeviceIdx, unsigned int Channel,
  unsigned int Threshold, unsigned int LowToHigh, unsigned int &Result) {


  unsigned char WaitStateProgram[8]; //buffer for WaitState program
  unsigned char *pWS = WaitStateProgram; //pointer that is needed during program formation
  unsigned char WaitStateReturn[4]; //buffer for returning packet

  int Status;

  if(LowToHigh) {
    //We first wait until AD converter result is below threshold
    //and then we wait till it is above it.
    Status = eProDas_WaitState_CMD_InternalAD(Channel, 0, Threshold-1, pWS);
    if(Status!=eProDas_Error_SUCCESS) { return Status; }
    Status = eProDas_WaitState_CMD_InternalAD(Channel, Threshold, 0, pWS);
  }
  else {
    //We first wait until AD converter result is above threshold
    //and then we wait till it is below it.
    Status = eProDas_WaitState_CMD_InternalAD(Channel, Threshold+1, 0, pWS);
    if(Status!=eProDas_Error_SUCCESS) { return Status; }
    Status = eProDas_WaitState_CMD_InternalAD(Channel, 0, Threshold, pWS);
  }
  if(Status!=eProDas_Error_SUCCESS) { return Status; }

  //until now the hardware has not been touched yet, only the WaitState program
  //has been written to the WaitStateProgram buffer

  //the following command now sends the program to the WaitState backbone,
  //which does the true waiting for a prescribed states in a proper sequence.

  Status = eProDas_WaitStateBackbone(DeviceIdx, WaitStateProgram, WaitStateReturn,
    unsigned int(pWS-WaitStateProgram), 2+2);
  if(Status!=eProDas_Error_SUCCESS) { return Status; }

  //collect AD result, which is spread amongn two separate bytes of returning packet
  Result = WaitStateReturn[2]+(WaitStateReturn[3]<<8);

  return eProDas_Error_SUCCESS;
}
```

# 24 The developers' set of functions

Functions in the developer set are not intended to be used by application developers on a regular basis. Many of these functions are not safe to use in a sense that they do not protect the developer from configuration and other mistakes. In the worst case scenario the improper use of these functions may destroy eProDas hardware or make other damages, like malfunction of the firmware.

## 24.1 Operations on the whole eProDas device

### 24.1.1 ReadDiagnosticInfo

Prototype in C/C++

```
int eProDas_ReadDiagnosticInfo(unsigned int DeviceIdx,
  eProDasStruct_DiagnosticInfo &Data, int ResetInfo);
```

Prototype in Delphi

```
function eProDas_ReadDiagnosticInfo(DeviceIdx: LongWord;
  var Data: TeProDasStruct_DiagnosticInfo;
  ResetInfo: integer):integer;
```

Prototype in VisualBasic

```
Function eProDas_ReadDiagnosticInfo(ByVal DeviceIdx As UInteger,
  ByRef data As TeProDasStruct_DiagnosticInfo, ByVal ResetInfo As Integer)
  As Integer
```

This function summarizes diagnostic information of the device. The configuration and operation of the device is not affected in any way, therefore the function is considered completely safe to use. Nonetheless, it is included in the developer set of functions since the retrieved information is generally not interesting for application developers. Instead, the obtained data primarily aids the hardware and firmware developers in hunting device bugs.

The retrieved data about the device is returned in the `Data` parameter, which is a structure that is described further on. You do not need to initialize any of the `Data` fields prior to calling the function. With a non-zero value of parameter `ResetInfo` you request that the non static part of information is to be reset or reinitialized after being retrieved.

For example, one of the returned pieces of information is `Errors` flags. This is a bit mask field, where each bit indicates whether certain error occurred or not. By instructing the device to reset the information all errors will be forgotten, which may be useful when monitoring the device to isolate condition or action that triggers certain error.

The definition of the `Data` structure is presented in the Table 24. Fields that are reset in the case of a non-zero parameter `ResetInfo` are denoted by an asterisk `(*)`.

Prototype in C/C++

```
struct eProDasStruct_DiagnosticInfo {
  unsigned int DeviceType;
  unsigned int DeviceTag;
  unsigned int FirmwareVersionLow;
  unsigned int FirmwareVersionHigh;
  unsigned int DLLVersionLow;
  unsigned int DLLVersionHigh;
  unsigned int KernelVersionLow;
  unsigned int KernelVersionHigh;
  unsigned int ResetType;  (*)
  unsigned int USBAddress;
  unsigned int UpgradeReport;  (*)
  unsigned int DebugIDC;
  unsigned int LowVoltageProgramming;
  unsigned int PinMCLREnabled;
  unsigned int CCP2PinMux;
  unsigned int Errors;  (*)
};
```

Prototype in Delphi

```
TeProDasStruct_DiagnosticInfo = record
  DeviceType: LongWord;
  DeviceTag: LongWord;
  FirmwareVersionLow: LongWord;
  FirmwareVersionHigh: LongWord;
  DLLVersionLow: LongWord;
  DLLVersionHigh: LongWord;
  KernelVersionLow: LongWord;
  KernelVersionHigh: LongWord;
  ResetType: LongWord;  (*)
  USBAddress: LongWord;
  UpgradeReport: LongWord;  (*)
  DebugIDC: LongWord;
  LowVoltageProgramming: LongWord;
  PinMCLREnabled: LongWord;
  CCP2PinMux: LongWord;
  Errors: LongWord;  (*)
end;
```

Prototype in VisualBasic

```
Public Structure TeProDasStruct_DiagnosticInfo
  Public DeviceType As UInteger
  Public DeviceTag As UInteger
  Public FirmwareVersionLow As UInteger
  Public FirmwareVersionHigh As UInteger
  Public DLLVersionLow As UInteger
  Public DLLVersionHigh As UInteger
  Public KernelVersionLow As UInteger
  Public KernelVersionHigh As UInteger
  Public ResetType As UInteger
  Public USBAddress As UInteger
  Public UpgradeReport As UInteger
  Public DebugIDC As UInteger
  Public LowVoltageProgramming As UInteger
  Public PinMCLREnabled As UInteger
  Public CCP2PinMux As UInteger
  Public Errors As UInteger
End Structure
```

**Table 24: Structure for retrieving diagnostic information.**

The fields `DeviceType` and `DeviceTag` hold the same values that would be retrieved by the `GetDeviceType` and `GetDeviceTag` functions, respectively (sections 21.3.1 and 21.3.2). In fact, these two functions internally call function `ReadDiagnosticInfo`, retrieve the appropriate value and discard the excessive information.

The fields `FirmwareVersionLow` and `FirmwareVersionHigh` return the reported firmware version of the device. `DLLVersionLow` and `DLLVersionHigh` report version of the library eProDas.DLL. Similarly, `KernelVersionLow` and `KernelVersionHigh` hold the reported version of the eProDas device driver. In each of the cases the related two numbers are displayed by the console and Delphi demo applications (chapters 8 and 9) in a form “*Item*`High.`*Item*`Low`”.

The `High` and `Low` parts of the version are considered a major and a minor version of the item, respectively. It is our intention to increase minor version number when small features are added or only bug fixes are applied to the code. However, major improvements to the code or cumulative additions of many small features would result in major version increase. The exact border between small and significant is newer well defined and we do not intend to concentrate thoroughly on this topic. The resolution of the issue is completely under the control of our intuition.

The field `ResetType` reports the last reset type that the device has experienced. The possible values are listed in the Table 25.

| Name | Value |
|---|---|
| `eProDas_Reset_PowerOn` | 0 |
| `eProDas_Reset_BrownOut` | 1 |
| `eProDas_Reset_MCLR` | 2 |
| `eProDas_Reset_MCLRIdleSleep` | 3 |
| `eProDas_Reset_WatchDog` | 4 |
| `eProDas_Reset_RESETInstruction` | 5 |
| `eProDas_Reset_StackOverflow` | 6 |
| `eProDas_Reset_StackUnderflow` | 7 |
| `eProDas_Reset_Unknown` | 666 |

**Table 25: Constants for reporting reset type.**

- `PowerOn` reset is a usual way of waking up the device from the state where it merely collects dust in your drawer.

- `BrownOut` reset indicates that the (`VDD`-`VSS`) voltage of the PIC microcontroller has fallen bellow the specified threshold (eProDas default is 4.3 V) for a short period of time. This may indicate that the power supply is not adequate and consequently the device is not working reliably. It can also mean that the blocking capacitors (see chapter 3) are inadequate. If you encounter this type of reset on a regular basis you are strongly encouraged to redesign the power supply of the device (or report the issue to us if we designed it :-).

- `MCLR` (master clear) indicates that an external circuitry requested reset by asserting low state on MCLR pin of the PIC. In order for this type of reset to work, the MCLR reset must be enabled through PIC configuration bits, which is not done by default in order not to lose pin RE3.

- `MCLRIdleSleep` (master clear during idle or sleep) is a variation of the previous error, where MCLR assertion happened when the device was in a sleep or idle mode. This difference may or may not mean something, depending on the situation.

- `WatchDog` reset in most cases indicates error in firmware since watch dog counter was not correctly reset. Currently, eProDas does not use the watch dog feature of the PIC so you can encounter this error only if you have extended the basic device functionality by yourself.

- `RESETInstruction` (reset by RESET instruction) is generally a bad thing, since firmware initiates this type of reset only if it encounters a fatal error from which it cannot recover. The exception is the firmware upgrade process, which finishes with reset instruction by design.

- `StackOverflow` and `StackUnderflow` indicate a nasty bug in firmware; if you code in assembler (or else), you know what does that mean. Please, report the issue to us.

- `Unknown` reset means that our routine for decoding reset causes does not cover all possible situations. Please, report the issue.

A Non-zero value of the `ResetInfo` parameter causes the `ResetType` to change to `PowerOn` after the information is retrieved.

If you are curious about what address has been assigned to your eProDas device by Windows, examine the field `USBAddress`. The possible addresses fall in a range from 1 to 127. This field has no practical meaning except for exploring the Windows algorithm for assigning USB addresses.

The field UpgradeReport is a bit field that indicates the state of the upgrade process. The meaning is summarized the Table 26.

| Bit Value | Meaning |
|---|---|
| 1 | Upgrade process has been started but it did not finish correctly. |
| 2 | Upgrade process has been started and completed successfully. |
| 4 | Internal upgrade structures are wrong. This is likely a firmware bug. |
| 256 | Write errors were encountered during upgrade process. |
| 512 | Write error indicator is wrong. This is probably a firmware bug. |

**Table 26: Interpretation of the UpgradeReport field.**

At the beginning of the firmware upgrade an “upgrade start” indicator is written in the data EEPROM of the PIC microcontroller; this indicator results in bit 0 (value of 1) of the field UpgradeReport being set. If you encounter this state ever in your life you are extremely lucky guy, since unsuccessful upgrade renders device unusable in almost every case, which means that this diagnostic information could not be retrieved in the first place.

When the upgrade finishes successfully the internal upgrade indicator changes into “upgrade finished”. This results in bit 1 (value of 2) of the UpgradeReport field being set. This is good news.

If upgrade indicator does not have a proper value, the bit 2 (value of 4) of the UpgradeReport is set. This is almost certainly a result of the bug in the firmware although it might also result from the malfunctioning of the data EEPROM.

Internally, firmware is upgraded in blocks of 64 bytes. When each block is written into the program FLASH memory of the PIC microcontroller, its contents is checked against the original contents. If mismatch is found a write error indicator is written into data EEPROM of the device and the block write is repeated. Bit 8 (value of 256) indicates that at least one write error was encountered during the upgrade process, which suggests that the FLASH program memory is approaching to the end of its lifecycle. If this indicator has an invalid value bit 9 (value of 512) is set in the UpgradeReport.

Upon each reset (other than the one that finished the upgrade process) all upgrade indicators are cleared. This means e.g. that if you unplug the device from USB after the upgrade and plug it back in, it will have no clue whatsoever about the upgrade and you will receive zero value of the UpgradeReport field.

A Non-zero value of the ResetInfo parameter causes the UpgradeReport field to be cleared after the information is retrieved.

A non-zero value of the DebugIcd field indicates that In-Circuit debugging is enabled. This also means that pins RB6, RB7 and RE3 are not available for general purpose usage.

A non-zero value of the LowVoltageProgramming indicates that a low voltage mode of programming the FLASH memory is enabled. This mode reserves pin RB5 for the task of firmware programming.

A non-zero value of the PinMCLREnabled indicates that reset of the device can be requested by asserting low state on pin MCLR. Otherwise, this pin may be utilized as RE3 digital input.

The fields DebugIcd, LowVoltageProgramming and PinMCLREnabled are examined by the function GetConfiguration to properly report roles of pins from RB5 to RB7 and RE3.

A non-zero value of the CCP2PinMux indicates that output pin for second PWM module is RC1. Otherwise this role is assigned to pin RB3.

The Errors field is a bit mask, where each set bit indicates that certain error was encountered. This field or its isolated bits are never cleared implicitly by the firmware, which means that the error may not necessary be encountered during the last operation but it could instead have appeared at any time in the past (the value is lost on power-off, of course).

The meaning of individual bits is briefly explained in the Table 27. If you are not a firmware or eProDas.DLL developer you do not need to be concerned with the meaning of the Errors field. But please, do report the errors to us if you encounter them.

| Bit | Meaning |
|---|---|
| 0 | Unknown low-priority interrupt (firmware error). |
| 1 | Unknown USB interrupt (firmware error). |
| 2 | USB packet with invalid PID received on endpoint 0 (maybe firmware error). |
| 3 | USB packet with invalid PID received on endpoint 1 (maybe firmware error). |
| 4 | USB packet with invalid PID received on endpoint 2 (maybe firmware error). |
| 5 | Invalid command parameter (eProDas.DLL error). |
| 6 | Invalid command packet size (eProDas.DLL error). |
| 7 | Invalid command (eProDas.DLL error). |
| 8 | Unknown high-priority interrupt (firmware error). |
| 9 | Periodic actions take too long (design flaw in interrupt routine?). |
| 10 | Invalid conditions for command (probably eProDas.DLL error). |
| 11 | unknown USB interrupt during periodic action (firmware error). |
| 12 | Reserved |
| 13 | Reserved |
| 14 | Program FLASH write error (probably hardware error). |
| 15 | Data EEPROM write error (probably hardware error). |

**Table 27: The meaning of bits in the Errors field.**

A Non-zero value of the ResetInfo parameter causes all Errors bits to be cleared after the information is retrieved.

### 24.1.2 UpgradeFirmware

Prototype in C/C++

```
int eProDas_UpgradeFirmware(unsigned int DeviceIdx, char *HEXFileName,
  unsigned int &TotalEraseRetries, unsigned int &TotalWriteRetries);
```

Prototype in Delphi

```
function eProDas_UpgradeFirmware(DeviceIdx: LongWord; HEXFileName: PChar;
  var TotalEraseRetries: LongWord; var TotalWriteRetries: LongWord):integer;
```

Prototype in VisualBasic

```
eProDas_UpgradeFirmware
```

This function is the heart of the UpgradeFirmware utility. It reads the new (strictly speaking it does not need to be new) version of the firmware from the file with the name HEXFileName and writes the contents to the program RAM of the PIC. The contents of the file must be in Intel HEX32 format or upgrade will be aborted with the one of the errors eProDas_Error_InvalidFileFormat or eProDas_Error_ReadFromFileFailed. If the file does not exist, the function will complain with an error eProDas_Error_OpenFileFailed.

Besides merely copying the contents of the input file over the old firmware this function goes to a great pain to make the upgrade process as safe as possible. Namely, if anything goes wrong during the upgrade process there is a chance that your device will not wake up ever again. With a rather elaborate procedure behind the scenes we tried to reduce the risk of a failure to a minimum. Hopefully, the device is in danger for 4 ms only, despite the fact that the upgrade process takes about 30 s to execute.

The details of the upgrade process are not described here. Instead, the interested reader is encouraged to examine the heavily commented source code of this routine.

The whole firmware in a specified file must not exceed the length 16384-2048 or the upgrade process will be (safely) aborted with an error `eProDas_Error_FirmwareTooBig`.

In parameters `TotalEraseRetries` and `TotalWriteRetries` the function reports potential erase or write errors, respectively, which are noticed during the upgrade process. Whenever an error is encountered the erase or write procedure is repeated a certain number of times (currently up to 3) to see whether the situation can be recovered. Any such repeated action increments the appropriate of the two counters. The non-zero resulting value in these two fields may indicate that the PIC's lifetime is approaching to the end. If erroneous operation does not succeed after 3 retries the upgrade process is aborted.

## 24.2 Direct access to various memories of the PIC

With this set of functions you may read from or write to program and data RAM of the PIC. Especially write operations expose the chip at your will and mercy. If you write nonsense data to a vital part of a memory you may temporally (until reset) or permanently (until you reprogram the device or re-solder the burned parts) destroy the functionality of the device.

### 24.2.1 ReadDataRAM

Prototype in C/C++

```
int eProDas_ReadDataRAM(unsigned int DeviceIdx, unsigned int StartAddress,
  unsigned int Length, unsigned char *Buffer);
```

Prototype in Delphi

```
type PByte = ^Byte;
function eProDas_ReadDataRAM(DeviceIdx: LongWord; StartAddress: LongWord;
  Length: LongWord; Buffer: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_ReadDataRAM
```

This function reads the `Length` number of bytes starting at `StartAddress` from PIC's data RAM into an array of bytes `Buffer`. If any part of the specified memory block fails outside the physically implemented memory, the function will return with `eProDas_Error_InvalidMemoryBlock` error code.

### 24.2.2 ReadProgramRAM

Prototype in C/C++

```
int eProDas_ReadProgramRAM(unsigned int DeviceIdx, unsigned int StartAddress,
  unsigned int Length, unsigned char *Buffer);
```

Prototype in Delphi

```
type PByte = ^Byte;
function eProDas_ReadProgramRAM(DeviceIdx: LongWord; StartAddress: LongWord;
  Length: LongWord; Buffer: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_ReadProgramRAM
```

This function works in a precisely the same way as function `ReadDataRAM`, except that it reads program RAM instead of data RAM.

### 24.2.3 WriteDataRAM

Prototype in C/C++

```
int eProDas_WriteDataRAM(unsigned int DeviceIdx, unsigned int StartAddress,
  unsigned int Length, unsigned char *Values);
```

Prototype in Delphi

```
type PByte = ^Byte;
function eProDas_WriteDataRAM(DeviceIdx: LongWord; StartAddress: LongWord;
  Length: LongWord; Values: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_WriteDataRAM
```

This function writes the `Length` number of bytes to PIC's data RAM, starting at address `StartAddress`. The values to be written are obtained from the array `Values`. If any part of the specified memory block fails outside the physically implemented memory, the function will return with `eProDas_Error_InvalidMemoryBlock` error code.

The same error is also returned if the requested length is greater than 40. When you need to write larger blocks, you need to call this function more than one time with properly specified parameters.

**Note.** Data RAM contains internal state of the device, its special function registers and USB buffers. Writing nonsense to these pieces of memory may result in various consequences.

### 24.2.4 WriteDataRAMWithBitMask

Prototype in C/C++

```
int eProDas_WriteDataRAMWithBitMask(unsigned int DeviceIdx,
  unsigned int StartAddress, unsigned int Length, unsigned char *AND_OR_Masks);
```

Prototype in Delphi

```
type PByte = ^Byte;
function eProDas_WriteDataRAMWithBitMask (DeviceIdx: LongWord;
  StartAddress: LongWord; Length: LongWord; AND_OR_Masks: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_WriteDataRAMWithBitMask
```

When working with configuration bits it is extremely useful to be able to just set or reset certain bit of a register without overwriting the entire value. This is what this function is about. Basically, it works in the same way as the function WriteDataRAM, except that you do not provide new values to be written to the target locations. Instead, for each target byte you provide the AND mask value and the OR mask value. The bits of the original value of the data RAM are first subject to bitwise AND operation with the AND mask value and then to bitwise OR operation with the OR mask value. The result is written back to the same location.

The length of the array `AND_OR_Masks` has to be twice the value of parameter `Length`, since for each influenced byte you need to provide two bit mask values. The first and the second bytes in the array are AND and OR masks for the first target byte, the third and the fourth bytes have the same role when manipulating the second byte, etc. The requested `Length` must not exceed 20 or the error `eProDas_Error_InvalidMemoryBlock` will result.

### 24.2.5 EraseProgramRAM

Prototype in C/C++

```
int eProDas_EraseProgramRAM(unsigned int DeviceIdx, unsigned int StartAddress,
  unsigned int Length);
```

Prototype in Delphi

```
type PByte = ^Byte;
function eProDas_EraseProgramRAM(DeviceIdx: LongWord; StartAddress: LongWord;
  Length: LongWord):integer;
```

Prototype in VisualBasic

```
eProDas_EraseProgramRAM
```

Before you can write to the program RAM, you need to erase it. In the opposite case only zero value bits will be written to the memory, since write operation cannot write ones. This is due to the principles of working of the FLASH program RAM. When the program RAM is erased it contains only ones and write operation becomes straightforward.

Due to the internal working of the PIC you can erase memory in chunks of 64-bytes only. The error `eProDas_Error_InvalidMemoryBlock` will arise if the parameter `Length` is not a multiple of 64. Also the `StartAddress` must be aligned on a 64-byte boundary in order not to trigger the same error.

To provide certain protection to the users, this function does not allow any part of the firmware to be erased. Specifically, if you try to erase memory below the address 16384-2048, you will get the error `eProDas_Error_InvalidMemoryBlock`.

### 24.2.6 WriteProgramRAM

Prototype in C/C++

```
int eProDas_WriteProgramRAM(unsigned int DeviceIdx, unsigned int StartAddress,
  unsigned int Length, unsigned char *Values);
```

Prototype in Delphi

```
type PByte = ^Byte;
function eProDas_WriteProgramRAM(DeviceIdx: LongWord; StartAddress: LongWord;
  Length: LongWord; Values: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_WriteProgramRAM
```

This function is a counterpart of the function `WriteDataRAM`, except that the values are written to the program RAM. Due to the internal working of the PIC the program RAM has to be written to in 32-byte chunks, so the `Length` must be a multiple of 32 or the now already boring error `eProDas_Error_InvalidMemoryBlock` will result.

To provide certain protection to the users, this function does not allow the firmware to be written over. Specifically, if you try to write to a memory below the address 16384-2048, you will get now the already well known error `eProDas_Error_InvalidMemoryBlock`.

**Note.** If memory block is not erased prior the writing operation, only zero value bits will be written to the memory. This is due to the principles of working of the FLASH program RAM.

### 24.2.7 ReadConfigurationByte

Prototype in C/C++

```
int eProDas_ReadConfigurationByte(unsigned int DeviceIdx,
  unsigned int ByteOffset, unsigned char &Result);
```

Prototype in Delphi

```
function eProDas_ReadConfigurationByte(DeviceIdx: LongWord; ByteOffset: LongWord;
  var Result: Byte):integer;
```

Prototype in VisualBasic

```
eProDas_ReadConfigurationByte
```

In addition to program RAM, data RAM and data EEPROM PIC microcontrollers also posses the so-called configuration bits, which hold a non-volatile configuration of the PIC. If you have a deep knowledge of PIC microcontrollers you may want to read these configuration bits in your application, for which you utilize function `ReadConfigurationByte`. This function reads one configuration byte at a time, which is specified by parameter `ByteOffset`, according to the Table 28.

The first column of the table specifies the value of the parameter `ByteOffset`, whereas the second column reveals the associated configuration octet (byte) as annotated by Microchip. Third and fourth column denote the physical address of the configuration data and predefined constant that you may use instead of plain number when specifying parameter `ByteOffset`, respectively. The last column indicates whether the value can be written to with the function `WriteConfigurationByte` in addition to be read with function the function `ReadConfigurationByte`.

| **ByteOffset** | **Microchip annotation** | **Actual Address (hex)** | **eProDas Constant** | **Writable** |
|---:|---|---|---|---|
| 0 | CONFIG1L | 300000 | eProDas_CONFIG1L | yes |
| 1 | CONFIG1H | 300001 | eProDas_CONFIG1H | yes |
| 2 | CONFIG2L | 300002 | eProDas_CONFIG2L | yes |
| 3 | CONFIG2H | 300003 | eProDas_CONFIG2H | yes |
| 5 | CONFIG3H | 300005 | eProDas_CONFIG3H | yes |
| 6 | CONFIG4L | 300006 | eProDas_CONFIG4L | yes |
| 8 | CONFIG5L | 300008 | eProDas_CONFIG5L | yes |
| 9 | CONFIG5H | 300009 | eProDas_CONFIG5H | yes |
| 10 | CONFIG6L | 30000A | eProDas_CONFIG6L | yes |
| 11 | CONFIG6H | 30000B | eProDas_CONFIG6H | yes |
| 12 | CONFIG7L | 30000C | eProDas_CONFIG7L | yes |
| 13 | CONFIG7H | 30000D | eProDas_CONFIG7H | yes |
| 254 | DEVID1 | 3FFFFE | eProDas_DEVID1 | no |
| 255 | DEVID2 | 3FFFFF | eProDas_DEVID2 | no |

**Table 28: Accessing configuration bits.**

If parameter `ByteOffset` does not contain one of the values in the first column of the table, the error `eProDas_Error_InvalidMemoryBlock` will result.

Upon successful operation the value of specified configuration octet is returned in parameter `Result`.

### 24.2.8 WriteConfigurationByte

Prototype in C/C++

```
int eProDas_WriteConfigurationByte(unsigned int DeviceIdx,
  unsigned int ByteOffset, unsigned char Value);
```

Prototype in Delphi

```
function eProDas_WriteConfigurationByte(DeviceIdx: LongWord; ByteOffset: LongWord;
  Value: Byte):integer;
```

Prototype in VisualBasic

```
eProDas_WriteConfigurationByte
```

This function, which is a counterpart of function ReadConfigurationByte, enables you to write new configuration information into non-volatile configuration bits. Again, parameter ByteOffset specifies the configuration octet to be written to (and is denoted as writable in the Table 28 or the error eProDas_Error_InvalidMemoryBlock will result). The new requested value is specified by the parameter Value.

**Important notes.**

Configuration bits are essential for proper start-up and working of the eProDas device. By writing improper or nonsense values your device will likely cease to work until you reprogram PIC microcontroller with dedicated hardware programmer. For example, if you specify usage of RC oscillator instead of quartz one, your device will not be able to communicate over USB bus any more. Consequently, your eProDas device will be unusable and you will not be able to send another WriteConfigurationByte command to repair the damage.

Like flash program RAM and data EEPROM, configuration bits are stored in a non-volatile type of memory that is subject to limited write/erase endurance. If you change the value of a configuration bit a couple of 10,000 times or so, you will eventually destroy the device. In order to make this scenario as unlikely as possible the function WriteConfigurationByte first reads the old value of configuration octet and initiates the actual write step only if the new requested value differs from the old one. This way it is safe to specify essential configuration values at the beginning of you application, which will not be written to non-volatile storage each time the application is executed. However, if you make a program loop where two different values are alternatively written to a certain configuration octet, you will destroy your chip in a short time.

## 24.3 Direct access to the “Access RAM” and Special Function Registers

Topologically speaking this material belongs to the previous section 24.2 about direct memory access. However, in the vast majority of cases, the memory that is accessed by eProDas.DLL and users’ applications belongs to the dedicated PIC’s memory that is called the “Access RAM”, which occupies the first 96 bytes of PIC’s data RAM bank 0 and the last 160 bytes of the bank 15.

**Note.** If you are not a hardcore PIC developer, the previous section certainly meant nothing to you and this one will look even more like a complete nonsense, so do not bother reading it.

Now, if you are still with us, then you certainly know that in the last portion of bank 15 there reside the so-called Special Function Register (SFRs), which are the link between software and hardware. For example, the already mentioned PORT, LATCH and TRIS registers are all located there along with registers for configuring AD converter, comparators, PWM, USART, SPI modules... The first portion of bank 0 is heavily utilized by eProDas for storing firmware variables.

To make the access to such important and frequently accessed memory chunk as efficient as possible there exist special functions for the task. These functions, which are the topic of this section, are a part of low-level portion of `eProDas.DLL` and other higher-level functions of the DLL rely on these functions to accomplish their hard work. For even more efficient but somewhat more tedious access to the memory in discussion, please see the section 24.4.

### 24.3.1 Specification of Access RAM addresses

The increased efficiency of the functions for accessing the PIC's Access RAM (by the way, we will try not to confuse you too much with the word "access" ☺) steams from the fact that the caller needs to supply only the lower portion of the address (8-bit value) and not the whole 16-bit number. Therefore, each time that Access RAM is manipulated by your application (directly or indirectly), one less byte needs to travel along the USB cable; further speedups are possible, though, as the section 24.4 reveals.

To emphasize the "compressed" address specification, the prototypes of functions within this group use the name `LowAddress` instead of `Address` or `StartAddress`, for the parameter that holds the requested memory address. If the value of the parameter `LowAddress` belongs to the interval from (0 to 95, inclusive), then the functions automatically interpret this as the memory that belongs to the bank 0 (firmware variables). On the other hand, `LowAddress`es from 160 to 255 (inclusively) are mapped to bank 15 and therefore point to SFR registers.

**Note.** The swapping from bank 0 to bank 15 never occurs in the middle of function progression. For example, if you specify to manipulate the memory block from addresses 90 to 100, all 11 bytes of reached memory would reside in bank 0, since the first specified byte resides there. None of the functions detects the crossing of border between address 95 and 96 internally, to switch the memory bank from 0 to 15. Fortunately, the need for such behaviour does not arise quite often, if at all.

### 24.3.2 ReadAccessSFR, WriteAccessSFR(WithBitMask),

Prototypes in C/C++

```
int eProDas_ReadAccessSFR(unsigned int DeviceIdx, unsigned char LowAddress,
  unsigned int Length, unsigned char *Buffer);

int eProDas_WriteAccessSFR(unsigned int DeviceIdx, unsigned char LowAddress,
  unsigned int Length, unsigned char *Values);

int eProDas_WriteAccessSFRWithBitMask(unsigned int DeviceIdx,
  unsigned char LowAddress, unsigned int Length, unsigned char *AND_OR_Masks);
```

Prototype in Delphi

```
function eProDas_ReadAccessSFR(DeviceIdx: LongWord; LowAddress: Byte;
  Length: LongWord; Buffer: PByte):integer;

function eProDas_WriteAccessSFR(DeviceIdx: LongWord; LowAddress: Byte;
  Length: LongWord; Values: PByte):integer;

function eProDas_WriteAccessSFRWithBitMask(DeviceIdx: LongWord; LowAddress: Byte;
  Length: LongWord; AND_OR_Masks: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_ReadAccessSFR

eProDas_WriteAccessSFR

WriteAccessSFRWithBitMask
```

These functions work in the same way as the functions `ReadDataRAM` (section 24.2.1), `WriteDataRAM` (section 24.2.3) and `WriteDataRAMWithBitMask` (section 24.2.4), respectively, do except that the memory block is specified according to the rules in the section 24.3.1.

### 24.3.3 AND, OR, XOR, XORXOR AccessSFR

Prototypes in C/C++

```
int eProDas_AND_AccessSFR(unsigned int DeviceIdx, unsigned char LowAddress,
  unsigned int Length, unsigned char *Values);

int eProDas_OR_AccessSFR(unsigned int DeviceIdx, unsigned char LowAddress,
  unsigned int Length, unsigned char *Values);

int eProDas_XOR_AccessSFR(unsigned int DeviceIdx, unsigned char LowAddress,
  unsigned int Length, unsigned char *Values);

int eProDas_XORXOR_AccessSFR(unsigned int DeviceIdx, unsigned char LowAddress,
  unsigned int Length, unsigned char *Values);
```

Prototype in Delphi

```
function eProDas_AND_AccessSFR(DeviceIdx: LongWord; LowAddress: Byte;
  Length: LongWord; Values: PByte):integer;

function eProDas_OR_AccessSFR(DeviceIdx: LongWord; LowAddress: Byte;
  Length: LongWord; Values: PByte):integer;

function eProDas_XOR_AccessSFR(DeviceIdx: LongWord; LowAddress: Byte;
  Length: LongWord; Values: PByte):integer;

function eProDas_XORXOR_AccessSFR(DeviceIdx: LongWord; LowAddress: Byte;
  Length: LongWord; Values: PByte):integer;
```

Prototype in VisualBasic

```
eProDas_AND_AccessSFR

eProDas_OR_AccessSFR

eProDas_XOR_AccessSFR

eProDas_XORXOR_AccessSFR
```

Frequently, you need to set, clear or toggle the state of some SFR bit(s), which you can accomplish with these functions; the function names should be self-descriptive. Functions with `AND`, `OR` and `XOR` in their names perform bitwise AND, OR and XOR operations, respectively, between the current value(s) of Access RAM and the user specified values in the array `Values`; the result is of course written back to the original location. The function …`XORXOR`… does a double toggling (about 83 ns apart) to produce equal resulting value as we started with, however with a forced transition; this function is a background workhorse of e.g. the functions `TogglePinTwice` (section 15.1.21) and `XORXOR_Port` (section 16.1.1).

### 24.3.4 ADD_AccessSFR

Prototypes in C/C++

```
int eProDas_ADD_AccessSFR(unsigned int DeviceIdx, unsigned char LowAddress,
  unsigned char Value);
```

Prototype in Delphi

```
function eProDas_ADD_AccessSFR(DeviceIdx: LongWord; LowAddress: Byte;
  Value: Byte):integer;
```

Prototype in VisualBasic

```
eProDas_ADD_AccessSFR
```

With this function you can add the `Value` to some location in Access RAM. Contrary to the previously described set of functions, here we do not specify the length of the memory block and we always operate on an isolated byte. The set of functions in the section 16.1.4 relies on the function `ADD_AccessSFR` to accomplish the task.

### 24.3.5 VarOperations

Prototypes in C/C++

```
int eProDas_VarOperations(unsigned int DeviceIdx, unsigned char LowAddress,
  unsigned char Flags);
```

Prototype in Delphi

```
function eProDas_VarOperations(DeviceIdx: LongWord; LowAddress: Byte;
  Flags: Byte):integer;
```

Prototype in VisualBasic

```
eProDas_VarOperations
```

This function differs substantially from other functions within the section 24.3 in a way that you do not provide a value of an operand for an operation to be performed on Access RAM location but you instead specify what actions to execute. These actions are specified by a bit field `Flags`, according to the following table.

| bit | bit weight | action |
|---|---|---|
| 0 | 1 | rotate left (no carry) |
| 1 | 2 | rotate right (no carry) |
| 2 | 4 | set or clear carry flag |
| 3 | 8 | rotate left (through carry) |
| 4 | 16 | rotate right (through carry) |
| 5 | 32 | one's complement |
| 6 | 64 | two's complement |
| 7 | 128 | swap nibbles |

**Table 29: Actions that the function `VarOperations` performs.**

Actions associated with bits 0 and 1 are obvious. The value of bit 2 is copied to the carry flag of PIC, by means of which the next two operations are influenced. By rotating the Access RAM value through a preset carry flag, you achieve logic shift operations. For example, to shift the value to the left and set it's bit 0, specify `Flags` as 12=8+4. The last three self-describing actions do not use the carry flag. Operations are executed in the order from bit 0 to 7, so the `Flags` value of 35=1+2+32, first rotates the value left, then it rotates it back right, and finally the one's complement of the value is performed.

**Note,** however that each operation is done directly on the Access RAM location in question. Therefore, the intermediate results influence the working of the eProDas device. For example, the transitions are visible on external pins if operations are executed on some port's latch register.

## 24.4 Dedicated "Access RAM" backbone

The Access RAM backbone is a special piece of eProDas firmware that enables you (or more likely us ☺) to perform the most frequent operations on Access RAM by transferring as little USB packets as possible between host PC and eProDas device; in addition the goal is to make these packets as short as possible for the sake of faster delivery.

When working with the backbone your application (or `eProDas.DLL`) specifies its demands by writing several commands along with potential parameters into one USB packet, which the backbone executes in a sequence. The results (if any) are written by the backbone into another USB packet to be delivered to the caller at the end of the execution of the whole "program" (USB packet contents). Therefore, in likely cases at most one USB packet travels in each direction to execute several operations on various Access RAM locations. More than one USB packet is needed only if not all commands can be stuffed into one packet (the limit of 44 bytes) or if intermediate results are needed by application for examination to determine further actions.

The backbone could also be described as a state machine with no more and no less than ten different states or modes of operation. These are: *read*, *write*, *AND*, *OR*, *XOR*, *read with increment*, *write with increment*, *AND with increment*, *OR with increment* and *XOR with increment*. Therefore, there are basically five different states, where each of them can have the optional tag *with increment*.

The modes are partially self-describing. In *read* mode the backbone reads certain Access RAM location and delivers the value to the host PC; more precisely, it writes the read value into USB buffer for later delivery. In *write* mode a specified value is written to the Access RAM location, whereas in *AND*, *OR* and *XOR* modes the Access RAM value is bitwise ANDed, ORed or XORed, respectively, with a specified value. Only the *read* and *read with increment* modes generate data to be delivered to the host PC and if the backbone does not executes any *read (with increment)* command, no returning USB packet travels from eProDas device to the host PC.

When in one of the incrementing modes the address of the Access RAM location is automatically incremented by the backbone and it needs not to be supplied for each operation through the USB cable, whereas in non-incrementing mode the user supplies the address for each action separately. For example, by exploiting the *write with increment* mode of operation it is possible to write the array of values into all five ports by specifying only the address of PortA latch register, the other four addresses need not occupy the USB packet.

### 24.4.1 The sequence of execution

The first byte of the "program" must contain the code (to be specified later) that tells the backbone into which of the above-mentioned modes you want it to start breathing. If the code specifies one of the incrementing modes, the next byte that follows must contain the number of repetitions of the command in this mode. After entering the prescribed mode, the backbone fetches the address of the Access RAM location (according to the section 24.3.1), the potential parameters and then it executes the operation that the mode determines.

In the case of non-incrementing command, the fetching of address and parameters repeats, which is followed by another execution of the same command (since we are still in the same mode) on a different memory location with different parameters.

In the case of incrementing operation, the fetching of parameters (but not the address) repeats, the previous address is incremented and the operation is executed again on an adjacent Access RAM location with new parameters (if you are familiar with workings of a burst RAM, you should have no trouble grasping the idea).

Here comes the peculiarity of the backbone. Whenever the incrementing mode of operation executes the command for the prescribed number of times, the backbone immediately switches to the non-incrementing mode of the same regime. For example, after repeating the *read with increment* of adjacent Access RAM locations for a given number of times, the backbone switches to the ordinary non-incrementing *read*.

Such behaviour saves USB bytes since there are not many situations where you need to execute certain action on different memory regions of the same length. Contrary, a typical situation is reading of all five ports and a couple of isolated SFR registers in a bunch. In such cases, there is no need to issue the separate command for switching from incrementing to non-incrementing mode, by means of which one precious byte is spared (yes, we do not live exactly in a poverty but we do appreciate such savings, nonetheless).

### 24.4.2 Transition of modes

Whenever the backbone works in a non-incrementing mode and it is ready to fetch the new address, it can also fetch the command for switching the mode of operation. The commands are differed from the addresses solely based on the values. The addresses from zero to ten are interpreted as commands for mode switching, according to the following table.

| code | meaning |
|---|---|
| 0 | exit & send potential *read* data to the host PC |
| 1 | switch to *read* mode |
| 2 | switch to *read with increment* mode |
| 3 | switch to *write* mode |
| 4 | switch to *write with increment* mode |
| 5 | switch to *AND* mode |
| 6 | switch to *AND with increment* mode |
| 7 | switch to *OR* mode |
| 8 | switch to *OR with increment* mode |
| 9 | switch to *XOR* mode |
| 10 | switch to *XOR with increment* mode |

**Table 30: Codes for switching modes of the Access RAM backbone.**

When the backbone fetches the new address and its value is greater than or equal to eleven, it is interpreted as memory address (section 24.3.1), so the backbone executes the appropriate action on it, according to the current mode of operation. If the fetched value is less than eleven, it is interpreted according to the Table 30 and the requested mode is entered. From now on, the new mode determines the action to be executed on the Access RAM locations that further addresses determine.

The *exit* command is optional, since the backbone can stop itself automatically at the end of the command USB packet.

If the code specifies a non-incrementing mode of operation (codes 1, 3, 5, 7 and 9), then the next byte that follows must be the next Access RAM address. Contrary, if the code specifies an incrementing mode of operation (codes 2, 4, 6, 8 and 10), then the next byte must specify the number of repetitions (from 1 upward; 0 is interpreted as 256 which does not have sense since USB packets cannot be that large) of the operation before the backbone switches to the non-incrementing mode. Then the next byte specifies the first Access RAM address in the interval.

As you can see, the first eleven Access RAM addresses are handicapped a bit since they cannot be reached directly. Fortunately, eProDas firmware is designed with this in mind so these addresses are occupied by firmware variables that are not of general interest to the application writer. The information held there is mostly of diagnostic value and it can be partially retrieved by the function `ReadDiagnosticInfo` (section 24.1.1).

If you really need to manipulate these obscure RAM locations, the backbone still enables you to do so. Namely, immediately after switching the mode of operation the backbone fetches the address of the next location without any further interpretation of the value. Therefore, to read the contents of the address 5, first switch to read mode and then immediately specify the address 5 as a target of operation. It is not an error if the backbone is already operating in *read* mode upon executing the switch. So, as you can see, we were thinking of everything ☺.

### 24.4.3 Detailed description of modes

In *read* mode each execution of reading only needs the address, so each further byte of the USB packet in this mode results in one Access RAM location to be read and the result put in the returning USB packet for delivery to the host.

In the *read with increment* mode the number of repetitions determines how many bytes are added to returning USB packet and only the starting address is provided.

In *write* mode the backbone needs the address and the value to be written to the RAM at the addressed location. Therefore, you need to provide two bytes of data for executing each write.

In *write with increment* mode you provide the starting address and then only the values (as many as there are repetitions) to be written to the adjacent RAM locations.

All other modes are similar to the *write* or *write with increment* mode. For plain non-incrementing mode of operation you always provide the pair (address, value) for each execution of the action, whereas in the incrementing mode you specify the starting address and as many values as there are repetitions.

### 24.4.4 Working with the Access RAM backbone

If you happen to have read the boring material about `Transfer` backbone (section 23.2) or `WaitState` backbone (section 23.3) then you know the drill. First, you build the USB packet that is stuffed with your commands, then you flush that packet down the eProDas toilet and then the eProDas system optionally throws another packet with results into the application's face (a scene from the middle of the movie DOGMA comes to our sick minds at this moment). Again, let us start with the function for sending an already prepared program (or USB packet) to the backbone.

#### 24.4.4.1 AccessSFR

Prototype in C/C++
```
int eProDas_AccessSFR(unsigned int DeviceIdx, unsigned char *OutPacket,
  unsigned int OutLength, unsigned char *InPacket, unsigned int InLength);
```

Prototype in Delphi
```
function eProDas_AccessSFR(DeviceIdx: LongWord; OutPacket: PByte;
  OutLength: LongWord; InPacket: PByte; InLength: LongWord):integer;
```

Prototype in VisualBasic
```
eProDas_AccessSFR
```

Call this function to send your already prepared program to the backbone for immediate execution. The parameter `OutPacket` points to a buffer with stored Access RAM program. Set the parameter `OutLength` to the length of your program (the length of the USB packet in bytes). Further, you need to set the parameter `InLength` to the exact number of bytes that are going to be returned by the backbone to you, and the parameter `InPacket` to the starting address of the buffer for the data.

If InLength is zero, then the function does not try to collect the data from the backbone and it demands that you specify 0 (null pointer) for the parameter InPacket. In the opposite case you must provide a valid address of the buffer or your application will crash.

**Note.** Take absolute care that the value of the parameter InLength equals the actual number of readings according to your program or various bad things will happen. If you specify a non-zero value for the parameter and there are no readings in your program, then eProDas.DLL will wait for the results forever. If you specify a zero value but there are some readings, then eProDas device will try to send the packet that eProDas.DLL will not collect and the USB pipe will come out of sync; eProDas will deadlock or crash in the future. Finally, if there are readings but their number does not match the non-zero value of InLength, then you will simply face the error eProDas_Error_InternalError.

One note that is worth remembering is that Access RAM programs (i.e. OutPacket buffers) remain intact and therefore they can be recycled. Your application can prepare programs in advance and then execute them repetitively as many times as needed without preparation overhead.

### 24.4.5 Building Access RAM program packets

(Copied & pasted from earlier backbones) As the first step allocate program buffer of capacity 42 bytes; even if your program is shorter, do not waste your precious time counting the number of needed bytes, since PC memory is cheaper than your labour. Even better approach is to allocate a slightly larger space (say 64 bytes) so that you can catch potential overflows without triggering memory violations or application crashes.

In addition, allocate one pointer variable (say, with name CmdBuffer) of the appropriate type (pointer to unsigned char in C/C++, pointer to byte in Delphi…) and initialize it to the beginning of the buffer. Now you are ready to utilize the following functions for filling in the packet.

When you are done, the difference between the current value of CmdBuffer and the starting address of the buffer is the length of the USB packet and consequently the value that should be supplied to the function AccessSFR through the OutLength parameter.

#### 24.4.5.1 AccessSFR_CMD_*Add*

Prototypes in C/C++

```
int eProDas_AccessSFR_CMD_AddAddress(unsigned char LowAddress,
  unsigned char *&CmdBuffer);

int eProDas_AccessSFR_CMD_AddData(unsigned char Data, unsigned char *&CmdBuffer);
```

Prototypes in Delphi

```
function eProDas_AccessSFR_CMD_AddAddress(LowAddress: Byte;
  var CmdBuffer: PByte):integer;

function eProDas_AccessSFR_CMD_AddData(Data: Byte; var CmdBuffer: PByte):integer;
```

Prototypes in VisualBasic

```
eProDas_AccessSFR_CMD_AddAddress

eProDas_AccessSFR_CMD_AddData
```

To add an address to the USB packet use the first listed function, whereas the second function comes in handy to add a data to the packet. In both cases the value of the parameter (LowAddress or Data) is added to the buffer directly and unchanged. The only difference between the two functions is that the first one refuses to work with LowAddresses that are smaller than 11 (section 24.4.2), whereas the second one will blindly take anything for granted.

**Note.** If you need access to the lowest and obscurest portion of Access RAM, i.e. below the address 11, you must use the second function to add that address to the packet. Remember, this is a valid and legal action only immediately after switching the mode of operation (section 24.4.2).

### 24.4.5.2 AccessSFR_CMD_*Switch*

Prototypes in C/C++

```
int eProDas_AccessSFR_CMD_SwitchRead(unsigned char *&CmdBuffer);

int eProDas_AccessSFR_CMD_SwitchWrite(unsigned char *&CmdBuffer);

int ePrODas_AccessSFR_CMD_SwitchAND(unsigned char *&CmdBuffer);

int eProDas_AccessSFR_CMD_SwitchOR(unsigned char *&CmdBuffer);

int eProDas_AccessSFR_CMD_SwitchXOR(unsigned char *&CmdBuffer);
```

Prototypes in Delphi

```
function eProDas_AccessSFR_CMD_SwitchRead(var CmdBuffer: PByte):integer;

function eProDas_AccessSFR_CMD_SwitchWrite(var CmdBuffer: PByte):integer;

function eProDas_AccessSFR_CMD_SwitchAND(var CmdBuffer: PByte):integer;

function eProDas_AccessSFR_CMD_SwitchOR(var CmdBuffer: PByte):integer;

function eProDas_AccessSFR_CMD_SwitchXOR(var CmdBuffer: PByte):integer;
```

Prototypes in VisualBasic

```
eProDas_AccessSFR_CMD_SwitchRead

eProDas_AccessSFR_CMD_SwitchWrite

eProDas_AccessSFR_CMD_SwitchAND

eProDas_AccessSFR_CMD_SwitchOR

eProDas_AccessSFR_CMD_SwitchXOR
```

When you want to switch the mode of backbone operation, use one of the `AccessSFR_CMD_`*`Switch`* or `AccessSFR_CMD_SwitchInc` functions. The former adds one byte to the packet (the code of new mode) whereas the latter in addition adds the number of repetitions as a second added byte.

**Note.** Change of operating mode replaces address fetching, so only insertion of these codes at the proper places (section 24.4.2) results in a desired action.

### 24.4.5.3 AccessSFR_CMD_*SwitchInc*

Prototypes in C/C++

```
int eProDas_AccessSFR_CMD_SwitchReadInc(unsigned int Repetitions, unsigned char
*&CmdBuffer);

int eProDas_AccessSFR_CMD_SwitchWriteInc(unsigned int Repetitions, unsigned char
*&CmdBuffer);

int eProDas_AccessSFR_CMD_SwitchANDInc(unsigned int Repetitions, unsigned char
*&CmdBuffer);

int eProDas_AccessSFR_CMD_SwitchORInc(unsigned int Repetitions, unsigned char
*&CmdBuffer);

int eProDas_AccessSFR_CMD_SwitchXORInc(unsigned int Repetitions, unsigned char
*&CmdBuffer);
```

Prototypes in Delphi

```
function eProDas_AccessSFR_CMD_SwitchReadInc(Repetitions: LongWord;
  var CmdBuffer: PByte):integer;

function eProDas_AccessSFR_CMD_SwitchWriteInc(Repetitions: LongWord;
  var CmdBuffer: PByte):integer;

function eProDas_AccessSFR_CMD_SwitchANDInc(Repetitions: LongWord;
  var CmdBuffer: PByte):integer;

function eProDas_AccessSFR_CMD_SwitchORInc(Repetitions: LongWord;
  var CmdBuffer: PByte):integer;

function eProDas_AccessSFR_CMD_SwitchXORInc(Repetitions: LongWord;
  var CmdBuffer: PByte):integer;
```

Prototypes in VisualBasic

```
eProDas_AccessSFR_CMD_SwitchReadInc

eProDas_AccessSFR_CMD_SwitchWriteInc

eProDas_AccessSFR_CMD_SwitchANDInc

eProDas_AccessSFR_CMD_SwitchORInc

eProDas_AccessSFR_CMD_SwitchXORInc
```

# 25 String functions

Many functions that are described in this section are not interesting to a broad programmer's audience. Instead, they are useful to writers of diagnostic tools or other related utilities. Namely, this set of functions converts certain subset of eProDas configuration data into descriptive strings that you may want to output to the screen or to a disk file. Both, the console utility (section 8) and Delphi demo application (section 9) heavily rely on these functions to provide user-readable information about eProDas device.

After reading the previous paragraph you are probably tempted to ignore this whole section altogether (this document is long and boring enough already) but anyway we would like to target your attention to functions that help you display frequency and similar information on screen, which are described in section 25.2.

## 25.1 Helpers of outputting configuration data

### 25.1.1 ErrorString

Prototype in C/C++

```
char* eProDas_ErrorString(int ErrorCode);
```

Prototype in Delphi

```
function eProDas_ErrorString(ErrorCode: integer): PChar;
```

Prototype in VisualBasic

```
eProDas_ErrorString
```

This function converts eProDas error code (Table 2) into a null-terminated hardwired string that equals the appropriate name of the constant without the part "`eProDas_Error_`". The individual words are separated with spaces, though.

In the case of an invalid `ErrorCode`, the string "Unknown Error Code" is returned.

### 25.1.2 TypeString

Prototype in C/C++

```
char* eProDas_TypeString(int TypeCode);
```

Prototype in Delphi

```
function eProDas_TypeString(TypeCode: integer): PChar;
```

Prototype in VisualBasic

```
eProDas_TypeString
```

This function converts eProDas device type code (**Error! Reference source not found.**) into a null-terminated hardwired string that equals the appropriate name of the constant without the part "`eProDas_Type_`".

In the case of an invalid `TypeCode`, the string "Unknown Device Type Code" is returned.

### 25.1.3 PinCfgString

Prototype in C/C++

```
void eProDas_PinCfgString(unsigned int PinConfig, unsigned int MinChars,
  char *StringBuffer);
```

Prototype in Delphi

```
procedure eProDas_PinCfgString(PinConfig: LongWord; MinChars: LongWord;
  StringBuffer: PChar);
```

Prototype in VisualBasic

```
eProDas_PinCfgString
```

This function synthesizes the string that describes pin roles according to the codes in the Table 22. Loosely speaking, each non-zero bit in the value `PinConfig` results in one added description to the resulting null-terminated string. Pin annotations are as follows.

- E    In combination with other markers denotes an error, associated with configuration of the pin. For example, if pin is configured both as digital output and input of AD converter, this marker would be displayed.

- –    Pin is not implemented so do not search for it on the pin diagram of PIC18F4550.

- *    Pin exists, but it is not available for you as a general purpose pin.
  For example, pin RA6 is used by quartz oscillator and pins RC4 and RC5 are dedicated to the USB connection, as described in the hardware section. If you are a lucky guy and you have an InCircuit Debugger, you will likely want to loose pins RB6, RB7 and RE3, which are needed for real-time debugging of the device. If you exploit this feature the three pins will also be notated with symbol *.

- DI    Digital input.
- DO    Digital output. Do not try to excite such pin with the external voltage supply.
- T    Pin is a clock source for one of the timer modules.
- P    Pin is an output of PWM module (it also has to be DO to actually generate signal).
- SO    Pin is output for SPI module.
- SI    Pin is input for SPI module.
- SC    Pin is clock for SPI module.
- UO    Pin is output of USART module.
- UI    Pin is input of USART module.
- AD    One of the multiplexed inputs of the AD converter.
- A+    Positive voltage reference of the AD converter.
- A–    Negative voltage reference of the AD converter.
- C+    Non-inverting input of voltage comparator.
- C–    Inverting input of voltage comparator.
- CO    Output of voltage comparator. Do not excite the pin with the external voltage supply.
- R+    Positive reference of voltage reference module (funny but realistic description).
- R–    Negative reference of voltage reference module (funny but realistic description).
- RO    Output of voltage reference. Do not excite the pin with the external voltage supply.

Since the result is synthesized rather than prepared in advance, you need to provide an array of characters with enough storage (typically, we allocate ample of space, since PC can easily cope with that), where the resulting string will be placed.

If the resulting string is shorter than the value of `MinChars` it is padded by enough spaces to becomes of a prescribed length.

### 25.1.4 ResetString

Prototype in C/C++

```
char* eProDas_ResetString(int ResetCode);
```

Prototype in Delphi

```
function eProDas_ResetString(ResetCode: integer): PChar;
```

Prototype in VisualBasic

```
eProDas_ResetString
```

With this function you can convert a code of the last reset type (Table 25) into a string. The function returns a pointer to a null-terminated hardwired string that equals to the appropriate name of the constant without the part “`eProDas_Reset_`”. For example, when specifying the `ResetCode` of 3 you get the string “`MCLRIdleSleep"`. If `ResetCode` is not valid you get a string “`Unknown Reset Code"`.

### 25.1.5 ErrorFlagsString

Prototype in C/C++

```
void eProDas_ErrorFlagsString(int ErrorCode, char *ResultBuffer);
```

Prototype in Delphi

```
procedure eProDas_ErrorFlagsString(ErrorCode: integer; ResultBuffer: PChar);
```

Prototype in VisualBasic

```
eProDas_ResetString
```

This function synthesizes the string that describes firmware errors according to the codes in the Table 27. Each non-zero bit among the lower 16-bits of `ErrorCode` results in one added error description to the resulting null-terminated string.

Since the result is synthesized rather than prepared in advance, you need to provide an array of characters with enough storage (typically, we allocate vastly too big amounts of spaces on the orders of couple kB, since PC can easily cope with that), where the resulting string will be placed.

## 25.2 Displaying frequency and other numerical data

### 25.2.1 Exponent2Prefix

Prototype in C/C++

```
Char *eProDas_Exponent2Prefix(int Exponent);
```

Prototype in Delphi

```
function eProDas_Exponent2Prefix(Exponent: integer): PChar;
```

Prototype in VisualBasic

```
eProDas_Exponent2Prefix
```

This function returns string that contains decimal prefix that is associated with the given exponent. Therefore, `Exponent` of 3 results in outputting letter “k” for kilo. Similarly, exponent of -9 returns string “n” for nano. If the value of `Exponent` does not represent an established prefix according to international system of units, the string “?” is returned. In the case of exponent -6, string “mu” for micro is returned, since Greek letters cannot be universally encoded in the plain 7 bit ASCII scheme, which is the only guaranteed set of universally available, displayable and printable characters. Yes, we could also use unicode but currently we do not, since not all development environments support this feature.

### 25.2.2 Double2HumanReadable

Prototype in C/C++

```
void eProDas_Double2HumanReadable(double &Number, int &Exponent);
```

Prototype in Delphi

```
procedure eProDas_Double2HumanReadable(var Number: double; var Exponent: integer);
```

Prototype in VisualBasic

```
eProDas_Double2HumanReadable
```

Whenever you want to display value of some physical quantity you may find this function handy. As the input you specify some value in parameter Number, which must be a variable and not a constant. The function adjusts this value to the range between 0 and 1,000. Parameter Exponent reveals for how much did the decimal point move within the process. For example, when Number equals 34,000,000 upon calling the function, the result is Number = 34, Exponent = 6.

As you have undoubtedly guessed this function is suitable for displaying numbers in human readable format. Instead of displaying ugly 34000000 g you can achieve more pleasant format of 34 Mg by using this function in combination with function Exponent2Prefix.

### 25.2.3 Frequency2String

Prototype in C/C++

```
void eProDas_Frequency2String(double Frequency, char *Result);
```

Prototype in Delphi

```
procedure eProDas_Frequency2String(Frequency: double; Result: PChar);
```

Prototype in VisualBasic

```
eProDas_Frequency2String
```

Frequency is probably the most often displayed physical value in data acquisition applications. This function helps you obtain string with a frequency value in human readable format. For example, the value 123456789 of parameter Frequency results in string “123.456789 MHz”. Please, make sure that the storage for resulting string, supplied by parameter Result, contains enough space for the generated string or your application will crash.

### 25.2.4 PrintFrequency

Prototype in C/C++

```
void eProDas_PrintFrequency(double Frequency);
```

Prototype in Delphi

```
procedure eProDas_PrintFrequency(Frequency: double);
```

Prototype in VisualBasic

```
eProDas_PrintFrequency
```

This function is usable to you only if you are developing textual (console) applications. It works the same way as function Frequency2String except that it does not output synthesized string to a buffer but it instead writes it to the console window (using C/C++ function printf).

# Appendices

# A Description of general purpose I/O ports and pins

Digital I/O (input/output) pins are a primary means of exchanging data between the eProDas device and other digital devices that you may connect to the PIC microcontroller. In the case of the PIC18F4550 chip digital pins are notated as pins RA0 to RA6, RB0 to RB7, RC0 to RC2, RC4 to RC7, RD0 to RD7 and RE0 to RE3. Please, examine the Figure 2 to see where these pins are physically located on the chip.

As far as microcontroller is concerned each I/O pin belongs to a certain PORT. For example, pins from RA0 to RA6 belong to the so called PORTA. Similarly, pins from RB0 to RB7 belong to PORTB, etc.

Whether a certain pin behaves as digital input or output is controlled by the TRIS register that is associated with the port. When certain bit of TRIS register is set (contains 1) the respective pin is digital input and the cleared TRIS bit (value of 0) turns the pin into a digital output pin.

For example, register TRISD contains binary value of $0011\ 0101_2$, which determines that pins RD0, RD2, RD4 and RD5 are digital inputs and pins RD1, RD3, RD6 and RD7 are digital outputs.

When pin is digital output it dictates (or at least it tries to do so in the limits of its current capability) the voltage on it as it is determined by the contents of the associated latch register, named LAT. A zero value of a bit in the LAT register instructs the associated pin to be in the state of logic 0, where it will output voltage of 0 V. Similarly, in the state of logic 1 the pin will output the VDD voltage, which in the case of an eProDas device means 5 V (nominal; the actual value varies since it is influenced AT LEAST with USB power supply voltage). Contrary to this behaviour, when pin is digital input it freely allows external circuitry to determine its voltage. The digital level of this voltage is available to you by reading the associated PORT register.

For example, register TRISD contains the same value as in the previous example. If you write the value 255, which is binary $1111\ 1111_2$ into the LATD register, the voltage of 5 V (nominal) will appear on pins RD1, RD3, RD6 and RD7, since these pins are configured as digital outputs and their respective bits in the LATD register contains 1. The value of LATD register has no influence on the remaining RDx pins, since these are digital inputs and their voltage level is determined by the outside circuitry and the binary voltage level can only be examined by reading the PORTD register.

When working with I/O ports and pins please note that not all of the PIC's pins are available to you. Namely, pin RA6 is dedicated to the workings of quartz oscillator and cannot be utilized as a general purpose pin. Similarly, pins RC4 and RC5 take care of USB communication exclusively. Further, if you are a hardware developer and you enable the In-Circuit debugging feature, pins RB6, RB7 and RE3 are reserved for the job. In a low voltage programming regime (which eProDas does not support but the brave developers among you may exploit anyway) you also lose pin RB5.

In addition, other peripheral modules of PIC such as AD converter and Voltage Comparators communicate with the outside world through the subset of the same device pins. Consequently, when you enable these modules you lose additional number of pins that take on their alternative role during the time that the respective modules are enabled.

Further, pay attention to the pins that are completely missing. Namely, there are no pins RA7, RC3 and RE4 to RE7. When reading values of ports with less than eight pins you still get the whole 8-bit result. Make sure that your application treats these phantom bit values as meaningless and it does not try to interpret them.

# B The acquisition time behind the scenes

The Figure 148 presents the analog model of internal AD channel, which has to be taken into account when designing AD system with prescribed accuracy.

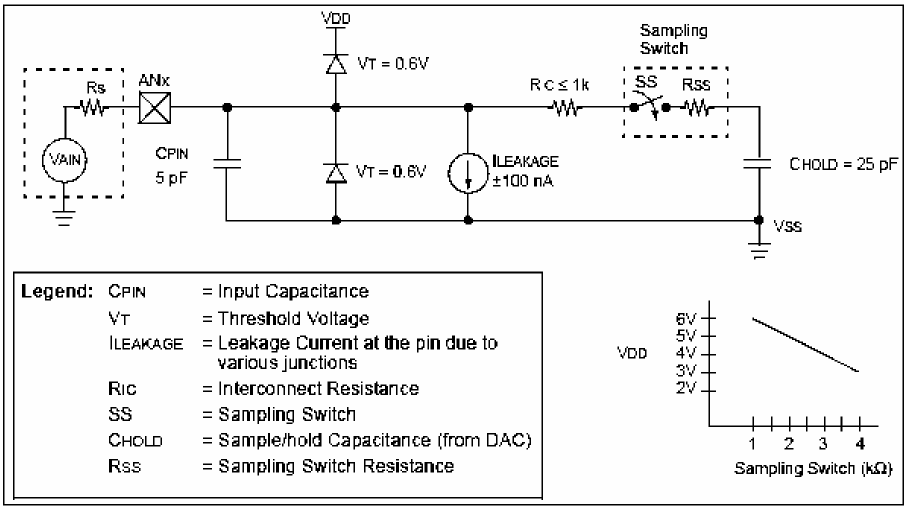

VDD
VT = 0.6V
Sampling Switch
RS
ANx
RC ≤ 1k
SS
RSS
VAIN
CPIN 5 pF
VT = 0.6V
ILEAKAGE ±100 nA
CHOLD = 25 pF
VSS
Legend: CPIN = Input Capacitance
VT = Threshold Voltage
ILEAKAGE = Leakage Current at the pin due to various junctions
RIC = Interconnect Resistance
SS = Sampling Switch
CHOLD = Sample/hold Capacitance (from DAC)
RSS = Sampling Switch Resistance
VDD
6V
5V
4V
3V
2V
1
2
3
4
Sampling Switch (kΩ)

**Figure 148: The analog model of PIC's AD channel.**

The capacitor CHOLD (capacitance of 25 pF) holds the input voltage during the AD conversion process, therefore the voltage on CHOLD is the voltage that is actually being converted when you instruct AD conversion and NOT the voltage on an input pin ANx. This is an important concept, which you should understand very well in order not to make big AD mistakes.

Namely, there are 13 analog inputs but only one AD converter within the PIC. When you switch a channel to be the target for the next AD conversion, you have to wait certain amount of time, the acquisition time, to allow the capacitor CHOLD to fully charge, before you start AD conversion. If you ignore this issue the capacitor will be charged only partially and you will get a completely wrong result of AD conversion.

Similarly, at the end of each conversion the capacitor gets discharged automatically and it has to be charged again before the new conversion starts. This is due to the fact that the internal PIC circuitry is optimized for one direction of voltage change (the charging) only and cannot cope with a falling voltage on a pin without cheating this way a bit. Therefore, even if you use only one AD channel all the time you still need to be concerned about the acquisition time.

Now, between the capacitor CHOLD and the voltage source VAIN, which you want to measure, there are some annoying but unavoidable resistances. Firstly, there is a resistance of the sampling switch RSS, for which you can deduce the value of about 2 kΩ from the graph in the lower right corner of the figure. Secondly, there is an interconnect resistance RC with a worst case estimated value of 1 kΩ. Thirdly, there is an internal resistance RS of your voltage source (which is under **your** control).

In every (non-pathological) case you want the sum of all these resistances to be as low as possible to make the process of charging the capacitor CHOLD as fast as possible. This is one of the reasons, why you usually hook operational amplifier, instrumentation amplifier of similar low impedance circuit to the front of the AD converter.

Please observe that with this approach you only make the resistance $R_C$ negligible. There is still an effective resistance of 3 kΩ along the analog path of your signal and the charging process cannot be completed instantly. Hence, regardless of what you do, there remains the need for properly setting the acquisition time.

For calculation of the required acquisition time you can find detailed formulas in the PIC datasheet, which we are not going to repeat here. Instead we will give you a hint. If the external resistance $R_S$ is not bigger than 2.5 kΩ then the acquisition time does not need to be longer than 6.4 µs; this result already includes the temperature coefficient and is valid up to the PIC's crystal temperature of 85 ºC.

By the way the 2.5 kΩ is the maximal recommended external resistance. Namely, as you can see in the Figure 148, there exists a parasitic or leakage current in a range of 100 nA. In a steady state this current flows entirely through the external resistance $R_S$, which results in a maximal voltage drop of 0.25 mV. In order to preserve the 10-bit accuracy, all errors arising from analog phenomena should be kept under half of the value of one LSB. With 5 V range this means ½ * (5 V) / 1024 ≈ 2.5 mV. Since there are many causes of errors, each one of them should be kept well below the stated maximum.

If the external resistance is significantly below 2.5 kΩ and/or you know that your PIC cannot heat close to 85 ºC it may be worth examining the formulas in the datasheet and calculate the actual required acquisition time, which may be significantly smaller than the stated 6.4 µs. Further gain may be achieved if you do not need 10-bit accuracy for which the result is calculated. The calculation assumes that capacitor $C_{HOLD}$ charges within the value of ½ LSB close to the input voltage. With smaller precision the value of ½ LSB increases and as a consequence the acquisition time shortens significantly.

# C Clarification of extended divider operations

It is obvious that the duration of one cycle of periodic actions must be less than the period of the periodic clock in order to prevent bumping of two consecutive periods together. The Figure 54 (page 166) illustrates the correct behaviour, whereby there is at least a small time gap between the end of each period and its respective successor.

**Example.** By selecting main divider of 2,400 (frequency of 5 kHz) the whole bunch of periodic actions (handling of USB transactions and execution of user program) must take less than 2,400 ticks of 12 MHz clock or 200 µs to complete.

When the extended divider is operational the circumstances are not so intuitive any more. Suppose that the user selects main divider of 60,000 and extended divider of 5. The length of the clock period is thus 60,000 times 5 ticks at 12 MHz, which equals 25 ms. Unfortunately, the permissive length of periodic program is radically shorter than this time interval.

The tighter timer requirements are due to the fact that only main divider is implemented in hardware but extended divider cannot be realized this way because PIC does not possess any 24-bit counters/timers neither it can logically cascade smaller hardware counters into a large one. The consequence is that whenever the period of main divider clock expires, PIC has to execute a small code fragment to implement the functionality of extended divider by means of incrementing 24-bit (3 times 8-bit) counter in software and checking the expiration of the period. Also, a limited handling of USB transactions is done at the same time in order to keep eProDas device responsive to user's request for terminating periodic actions. All in all, 18 ticks at 12 MHz are spent during this time. The situation is illustrated in the following figure.

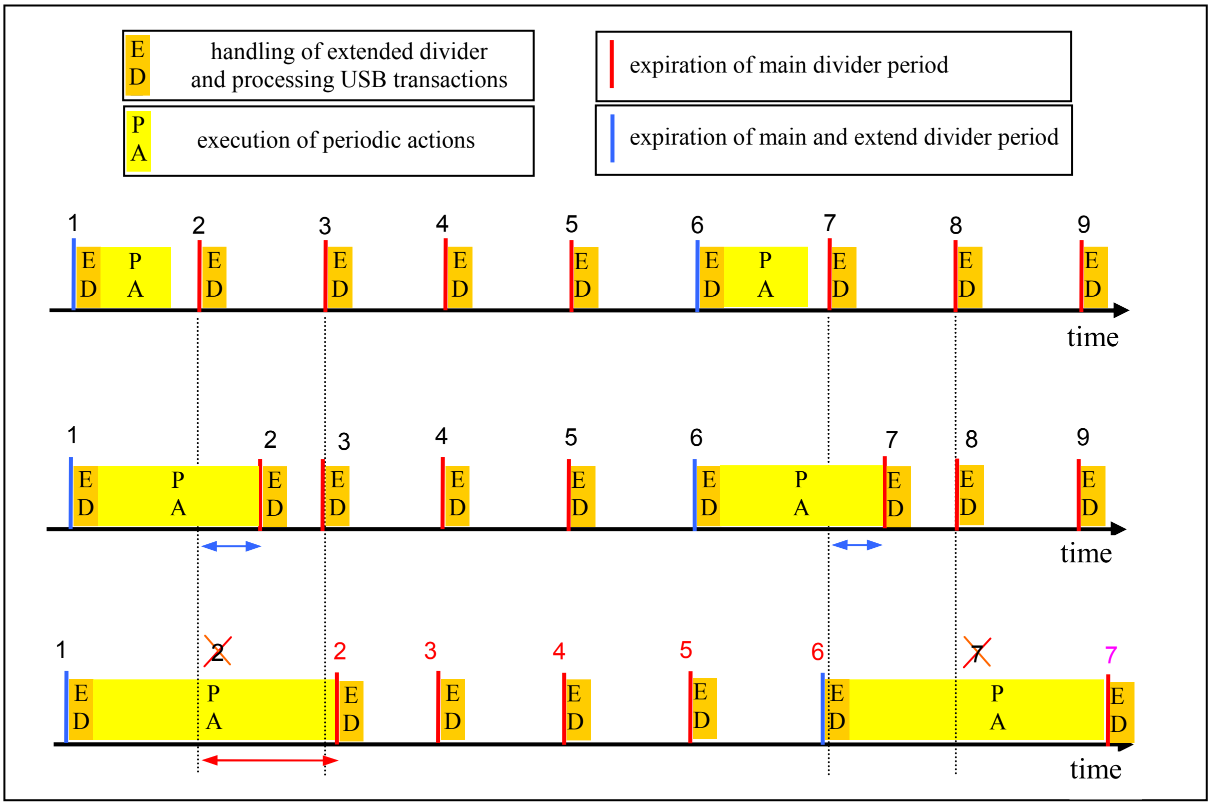

ED handling of extended divider and processing USB transactions
PA execution of periodic actions
expiration of main divider period
expiration of main and extend divider period
1 2 3 4 5 6 7 8 9
ED PA ED ED ED ED ED PA ED ED ED
time
1 2 3 4 5 6 7 8 9
ED PA ED ED ED ED ED PA ED ED ED
time
1 2 2 3 4 5 6 7 7
ED PA ED ED ED ED ED PA ED
time

**Figure 149: Execution time limit when extended divider is operational.**

Let's take a look at the first row. Main divider period expiration is denoted with red and blue vertical bars. Sticking with the previously selected main divider value this happens once per 60,000 ticks at 12 MHz or once per 5 ms. At each such occurrence 18 ticks are spent on handling the extended divider and USB transactions (orange rectangles with letters ED). If extended divider period does not expire (like on occasions 2, 3, 4, 5, 7, 8 and 9), there the process stops and PIC merely waits for the next such occurrence. However, when the extended period does expire (like on occasions 1 and 6) PIC starts to execute user's periodic program in a usual way (yellow rectangles with letters PA).

The top row presents the situation where total execution time of all described activities takes less than the length of the main divider period. In such cases everything executes normally and no special explanation is necessary. The second row illustrates a more subtle example. Here the duration of activities is longer than the length of main divider period. As we can see, processing of extended divider on occasion 2 is delayed, since previous periodic actions extend into the time interval that belongs to ED activities (event 2).

Still, no harm has been done, since the delay is short enough that extended divider processing can be completed before the occurrence 3. Namely, PIC's hardware internally buffers one main divider expiration event and executes the handling routine as soon as it can. By examining the second row we see that the next periodic actions at occurrence 6 are still executed on time. Neither any jitter has been introduced nor the period (frequency) of execution changed.

The third row reveals a disaster. Periodic actions not only take more time than one main divider period to execute but they take more time than the length of the two such periods. The processing of true occurrence 2 is delayed for so long that the true occurrence 3 happens before the processing of the former can take place. PIC cannot buffer more than one request for processing and consequently occurrence 3 (or 2, whichever you prefer) is silently discarded. As the third row demonstrates, one main divider period is lost and cannot be recovered. From now on, the extended divider constantly lags behind for that amount of time.

Due to the described unfortunate timing of events, the next periodic actions happen one main divider period later than planed (at true occurrence 7 instead of 6). Periodic actions at fake occurrence 6 (true 7) are again executing for too long and the loss of main divider period repeats. The bottom line is that the frequency of periodic actions differs from the configured one and such situations should be avoided if precise execution times are important.

However, if you can tolerate the described execution anomalies you can well go along with it. In the cases where eProDas must to a lot of work in one period of execution, this is the only choice. Also, in the majority of situations (but not always) the execution will be jitterless and with stable and repetitive frequency. The only annoyance is that the actual frequency of execution is not the same as the demanded one. By taking a little effort you can even calculate the actual frequency by taking into account dropped extended divider's periods; it is perfectly predictable but tedious to calculate. We might teach eProDas how to calculate these figures somewhere in the future, but currently we do not have the time to do it.

When you do not want to do the math by yourself and you need to set the frequency of execution in a usual way, you have no choice but obey the following rule: periodic actions (PA) plus two times processing of extended divider and USB transactions (ED) must take less time than two periods of the main divider.

**Example.** Main divider is 60,000. According to the rule we have (2 x 60,000)–(2 x 18) = 119,982 ticks for our periodic program. The permissive length is thus slightly less than 10 ms (9.9985 ms to be exact) and not 25 ms as one might expected.

Although all this scares you, do not worry. The function `AnalyzePeriodicActions` does the math for you so just stick with us and give it a try.

**Note 1.** According to the previous description we conclude that permissive length of periodic program is determined solely by main divider. Extended divider does not have any influence whatsoever.

**Note 2.** Always select dividers in such a way that main divider is as large as possible for obtaining the desired frequency. For example, selecting main divider of 15,000 and extended divider of 100 results in frequency of 8 Hz. The choice of dividers is not optimal, since the same frequency could obviously be obtained by selecting main divider of 60,000 and extended divider of 25.

Selecting dividers in accordance with this rule is needed or recommended for maximizing the time that your periodic program is permitted to execute without bumping into the code for extended divider handling.