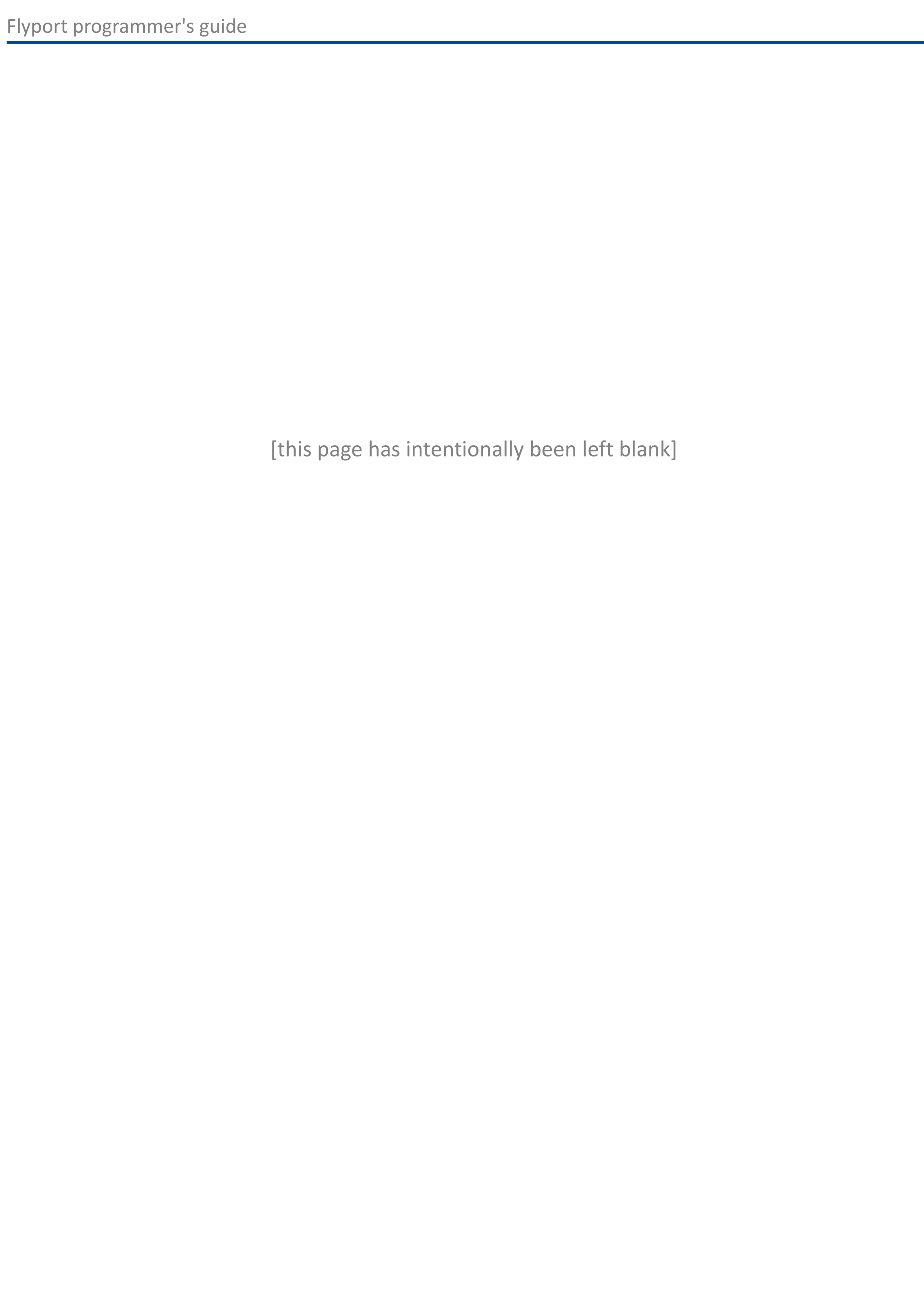
Flyport Programmer's Guide

Framework version 2.1

release 1.0



[this page has intentionally been left blank]

Contents

Flyport overview....5
- Flyport hardware....5
- Flyport bootloader....6
  - The serial bootloader....6

Controlling the Flyport hardware....7
- Digital Inputs and outputs....7
  - Digital I/Os functions....8
- Remappable pins....11
  - Remappable pins functions....13
- Analog inputs....14
  - Analog inputs functions....15
- PWMs....16
  - Using PWM with Flyport....17
- Serial communication (UART)....19
  - UART functions....20
- I2C communication protocol....22
  - I2C functions....22
  - I2C functions usage library example....23

Using the TCP/IP stack....25
- Managing the Wi-Fi....25
  - The connection profiles....25
  - Connection functions....27
  - Customizing network parameters at runtime....30
  - Wi-Fi functions and variables....32
- TCP Protocol....34
  - TCP Functions....34
  - TCP usage....36
- UDP Protocol....39
  - UDP Functions....39
  - UDP usage example....42
- SMTP Protocol....43
- FTP Client....45
- The webserver and HTTPApp.c....46
  - What is a webserver and how it works....46
  - Flyport webserver and how it works....46
  - Dynamic variables....47
  - AJAX in action....52
- SNTP Client....55
  - SNTP functionalities....55

SNTP usage example....56
Advanced Features....59
RTCC peripheral module....59
RTCC structure....59
RTCC functions....60
The energy saving modes....62
Hibernate mode....62
Sleep mode....63
Energy saving usage example....64

Flyport overview

Flyport hardware

Flyport is a wireless device embedding a **Microchip PIC24FJ** microcontroller and a **Wi-Fi transceiver**. It has a 26 pin connector to communicate with external electronics, and can be powered with **3.3 or 5V**.
Flyport is a standalone system, it embeds the **TCP/IP stack** to control the Wi-Fi and can be programmed with the custom firmware of the user to accomplish many actions. Like controlling relays, reading digital and analog IOs, communicating with UART, I2C or SPI buses and so on. The PIC24 is a **16 bit, 16 MIPS** microcontroller, with **256 KB flash memory and 16 KB of RAM**. The Wi-Fi module can connect to 802.11 b/g/n networks. Flyport only needs power supply and it can be used a **web server** (with HTML pages and AJAX components), **SMTP client** to send email**,** remote **TCP or UDP client/server** and much more.
To program the Flyport it can be used a USB "nest", that can be connected to the standard USB port of the PC. After installing the drivers, the Nest is seen as a serial port, useful to program the device and to debug the firmware.

Fig. 1 – Flyport and USB nest

Flyport bootloader

Flyport uses a serial bootloader, a real time operating system, the TCP/IP stack, the OpenPicus Framework, and the custom user firmware.

The serial bootloader

QUESTION: what is a serial bootloader and why is it needed on an embedded device like Flyport?
Well, to program a microcontroller, usually is needed **a serial programmer**, an external device which writes on the memory of the microcontroller the new firmware and controls the boot and the reset of the device. The programmer is connected to the PC that sends the firmware and the programmer writes inside the microcontroller flash memory.
Instead, using the internal bootloader, it's possible to program the microcontroller just using a serial connection, for example the USB-to-serial, like in USB nest, or just an UART connection.
The bootloader is a small program that starts anytime the microcontroller boots, and awaits on the serial port for a special message for some second. When it receives this special message (usually a string) it “understand” that the IDE is trying to program the micro, so it reads the commands arriving on the serial port and writes them inside the microcontroller memory using the RTSP – real time serial programming - technique.
So **the micro can writes its own memory without any external device**! Easier and cheaper.

QUESTION: the bootloader is located inside the program memory, and it writes inside the program memory. This can be dangerous? What happens if the bootloader tries to “overwrite itself”?!
When the PC sends some instruction to write in a “dangerous” memory address, the bootloader stops writing, avoiding “killing” itself. The IDE gives a feedback to user, saying “the code can damage the bootloader, so it has not written it”.

QUESTION: so the bootloader is another program resident inside the microcontroller. Will it slow the micro? And will it reduce the available memory for the user firmware?
The bootloader is executed **only at the startup** of the Flyport, so it doesn't slow down in any way the Flyport. It uses very few memory, the Flyport has 256KB of memory, and the bootloader keeps just 1KB! So no real reduction for the user's firmware.

NOTE: *the Flyport uses a slightly customized version of the ds30 bootloader. An opensource and lightweight bootloader for PIC microcontrollers*
(http://mrmackey.no-ip.org/elektronik/ds30loader/).

Controlling the Flyport hardware

In this chapter will be shown how to control the hardware of Flyport, the digital IOs, the analog inputs, how to create PWM and how communicate with other devices or peripheral.

QUESTION: usually, to control the hardware of an embedded device is required to know the registers and change them. Must I know the registers and the hardware of the microcontroller?
No, the OpenPicus Framework gives you a set of instructions to control the hardware of Flyport without knowing the microcontroller hardware or all the registers.

Digital Inputs and outputs

Flyport has 26 pins, in the **webserver default configuration** there are **5 inputs and 5 outputs**. It is just a “startup” configuration, but **it can be changed any time is needed**. So any pin can be configured as digital input or digital output. The other pins are dedicated to specific peripheral functions, but can be used also as digital input or digital output.

QUESTION: how are named the Flyport's pins?
There are two ways to refer to the Flyport pins: p1-pn, where the number refers to the related number of the pinstrip, and d1in-d5in and d1out-d5out, which refers to the standard pinout configuration. Clearly, if you want to modify the pinout configuration, will be easier to use the “generic” names p1-pn.

NOTE: *The* ***dxin/dxout is a deprecated notation****, and user should prefer the using of p1-pn notation, since it is the most flexible and useful for remappable pin configuration. Using the name d3in for a PWM output could be very uncomfortable. In the future releases there could be no dxin/dxout notation, and user should no more use them.*

See an example:

```
#include "taskFlyport.h"

void FlyportTask()
{
	IOInit(p5,out); //Init the p5 pin to digital output function
	IOInit(d2out,in); //the d2out is digital in
	while (1)
	{
		// MAIN LOOP
	}
}
```

Taking a look at the program we see that in the first case is used the name **p5** to access one pin, while in the second IOInit command is used the name **d2out**.
It can be used both the names of the pin by standard function or pin number, so

```
	IOInit(p5,out);
```

and

```
	IOInit(d1in, out);
```

are the same commands, written in two different ways.

NOTE: *We suggest to use ONLY pin numbers, since dxin/dxout are deprecated*

Digital I/Os functions

To change the state of a Digital Output → **IOPut**(pin name, value);

For example:

```
IOPut(p6, on);          //sets the pin to high voltage value (3,3V)
IOPut(p6, off);         //sets the pin to low voltage value (0V)
IOPut(p6, toggle);      //changes the state of the pin from high-to-low or low-to-high
```

The “on” keyword is associated to a high voltage level, so a “TRUE” logical state.
In a similar way the “off” keyword is associated to a low voltage level, so a “FALSE” logical state.

To read the state of a Digital Input → **IOGet**(pin name);

For example:

```
    IOGet(d1in);        //will return the value of the pin - on(1) or off(0)
    IOGet(p5);          //same function above, but using Pin Number
```

To initialize a Digital Input or a Digital Output → **IOInit**(pin name, type);

By default the digital inputs/outputs are initialized accordingly to their name (for example d1out is already setted as output).
It could be necessary to reconfigure a digital I/O by the user application in a different function. To do this, we can use the IOInit function.

For example:

Output pin → **IOInit(p5, out);**
Input pin → **IOInit(p6,in);**

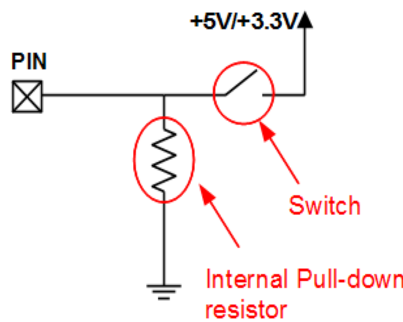
For inputs, we can set also 2 more settings:

Input pin with internal pull-up resistor → **IOInit(p5, inup);**
Input pin with internal pull-down resistor → **IOInit(p5, indown);**

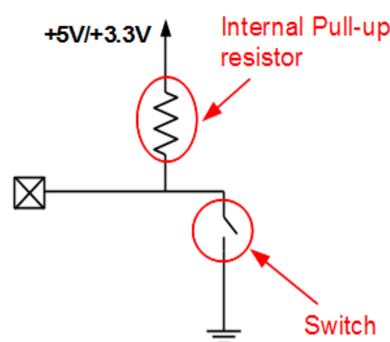
QUESTION: What does it mean pull-up and pull-down resistors in the Flyport's pins?
Pull-up and pull-down resistors are used to avoid floating voltages on input pins. With pull-up resistor we connect a input pin to a high voltage reference, instead with pull-down resistor we connect it to a low voltage reference. Anyway you can always change the input value with another voltage source, or with a switch, as shown in figure:

Pull-up and pull-down resistors

Pull-down:



Pull-up:



Switch closed: `IOGet(pin) = ON`
Switch open: `IOGet(pin) = OFF`

Switch closed: `IOGet(pin) = OFF`
Switch open: `IOGet(pin) = ON`

In the pull-down circuit of the figure, we can see that when the switch is opened (no other sources are connected), the input pin can "read" a low voltage value. If we close the switch (and connect a different voltage source), we change the voltage reference to a high value.
In the pull-up case, we have a high value when the switch is open, and a low value when the switch is closed, because the internal reference is high and the external reference is low.
Another convenience of using the internal pull-up/pull-down resistors is that them are placed inside the microcontroller, and you can change them without external components.

NOTE: *Pay attention of the different pull-up/pull-down values on switch states!*

QUESTION: How can we catch a pushbutton state change? Pressed or Released?
Buttons ALWAYS need internal pull-up ("inup") or pull-down ("indown") resistors.

Input type	Button pressed	Button released
inup	ON to OFF	OFF to ON
indown	OFF to ON	ON to OFF

To **Check** the state of a pushbutton → **`IOButtonState(`*`pin name`*`)`**
Returns: ***pressed*** *if the button has been pressed*
released *if the button has been not pressed or released*

You don't have to keep track of the state of the pin, or of its logical value. The Framework does this work for you and tells you if the button has been pressed or released.

For example:

```
if(IOButtonState (p5) == pressed)
{
	// Code to do when p5 is pressed...
}
else
{
	// Code to do when p5 is not pressed...
}
```

To know what kind of value you have to put instead of pressed, you have to check the table previously showed.
In the case of “inup” resistor you have to put “OFF” (that is a low voltage level).
In the case of “indown” resistor you have to put “ON” (that is a high voltage level).

A frequent problem related to the catch of button pressing state is the **bouncing** of the signal.
This problem is generated by mechanical issues on the internal contacts of buttons but with a small software attention can be solved well.
The **IOButtonState has also a de-bounce feature integrated**, so you don't have to worry about bad contacts of buttons, or false spikes in input signal caused by mechanical problems inside the button itself that could give wrong informations of real button pressing state.

The results will be filtered on 20ms of time:

- if the input value changes inside a time of 20ms, the result will not be valid
- if the input value remains the same inside a time of 20ms, the result will be valid

Remappable pins

A great feature of Flyport is the ability of remap the peripheral modules to almost any pin.

Flyport PIN	NAME	Function	Remappable
1	SCL	I2C	NO
2	IN5		YES
3	SDA	I2C	NO
4	OUT1		YES
5	IN1		YES
6	OUT2		YES
7	IN2	INT0	INPUT ONLY
8	SCLK		YES
9	IN3		YES
10	SDO		YES
11	IN4		YES
12	SDI		YES
13	URX		YES
14	CS		YES
15	UTX		YES
16	+5V		NO
17	OUT3		YES
18	PGC AN4		YES
19	OUT4		YES
20	PGD AN3		YES
21	OUT5		NO
22	GND		NO
23	AN1		YES
24	+3V		NO
25	AN2		YES
26	MCLR		NO

Remappable inputs and outputs table

As you can see on the table, almost all the Flyport pins are usable as remappable pins. The only un-remappable pins on microcontroller are: I2C pins SDA & SCL, and OUT5. There is also the Input Remappable pin 45, the p7 pin of Flyport's pin strip, that can be used for input functionalities only in the remapped peripheral.
So, if you want to use the remapping feature, you can associate PWMs, SPI2, UARTs TIMER4, and EXTERNAL INTERRUPTs to the pins that can use this feature.

For PWMs you can use the PWM dedicated functions (see PWM section), for the other peripheral here is a list of functionalities you can associate to every pin.

QUESTION: What are the various assignable Functionalities at Remappable Pins? How are them named?

Output peripherals

- UART1TX
- UART1RTS (not enabled in default UART initialization)
- UART2TX
- UART2RTS (not enabled in default UART initialization)
- UART3TX
- UART3RTS (not enabled in default UART initialization)
- UART4TX
- UART4RTS (not enabled in default UART initialization)
- SPICLKOUT (for SPI Master mode, Clock Output Signal)
- SPI_OUT (Data Output Signal)
- SPI_SS_OUT (for SPI Master mode, Slave Select Signal)

Input peripherals

- UART1RX
- UART1CTS (not enabled in default UART initialization)
- UART2RX
- UART2CTS (not enabled in default UART initialization)
- UART3RX
- UART3CTS (not enabled in default UART initialization)
- UART4RX
- UART4CTS (not enabled in default UART initialization)
- EXT_INT2
- EXT_INT3
- EXT_INT4
- SPICLKIN (for SPI Slave mode, Clock Input Signal)
- SPI_IN (Data Input Signal)
- SPI_SS (for SPI Slave mode, Slave Select Signal)
- TIM_4_CLK

With the Flyport module pinstrip connector it can be created more expansions with different pinouts just using remapping feature. As a result all the layouts of Flyport expansion boards can be more simple and easy to route as well as on pcbs, breadboards or any prototyping board type.

Those configurations are very dependant on user specific application, so every application should have got a *"Hardware Architecture"* that should be decided first to know how to use peripherals, and what kind of pins are "simple I/Os"

Remappable pins functions

To REMAP a pin you can simply use the IOInit function. The difference between the digital I/Os and PeripheralPinSelect assignment is done with different values of the second parameter:

IOInit (p2, EXT_INT3); will associate the pin 2 of Flyport to the External Interrupt 3 functions.

IOInit (p18, SPI_OUT); will associate the pin 18 of Flyport to the SPI2 data out functionality.

IOInit (p7, SPI_OUT); *WILL NOT WORK* because pin 7 can be only a input peripheral pin select!

Note: *Those functions are useful, but a little attention should be done. If a pin is assigned to a peripheral, the IOGet and IOPut will not work properly. Secondly, after the assignment, a new function will be associated to a pin, and it must be used the related peripheral functions to set up the peripheral module.*

For example the UART2, UART3 and UART4 are *not enabled by default* to give more memory to user application. If it is planned to use 3 UARTS on the user application, them must be enabled in the Openpicus IDE Wizard. See UART module for more informations.

QUESTION: Can I use the remapping feature at runtime?
Openpicus Framework can afford also runtime changing of pins mapping but it should be done with some attention, specially in the layout design since connect different type of peripheral modules can damage the hardware itself. **We suggest to first plan a right pin mapping, apply it on Flyport's first operations, and leave it as is for the rest of the application.**

Analog inputs

Flyport has 4 analog pins, named A1_in, A2_in, A3_in and A4_in. Like their name say, are capable of read analog voltages. With those pins we don't have only 2 values like digital I/Os, but values that can start from 0 to 1023 because of the hardware structure of the internal ADCs of the microcontroller. In fact, with a 10bit conversion, we can read 2^10 = 1024 different values, so from 0 (min voltage readable) to 1023 (max voltage readable).

QUESTION: What is the relation between number and voltage?
In terms of voltages, Flyport uses on-board regulated reference of 2.048V for max voltage input, so your analog source should be inside this range. If an analog signal will be >2.048V (but less than 3.3V) it will be read as 1023, if <0V it will be read as 0. So, the relation between the number and the real voltage input is:

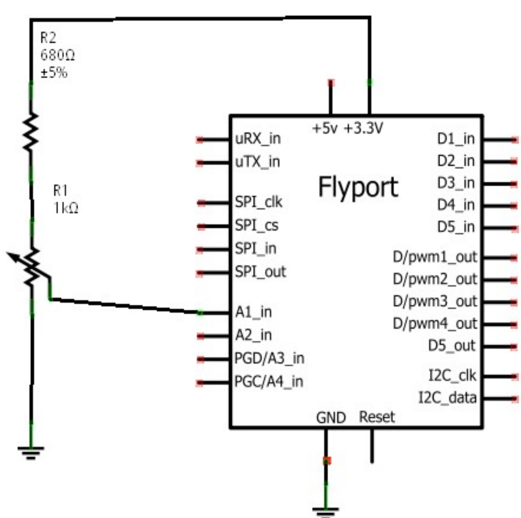
2.048V / 1024 = 0.002 V = 2mV minimum step

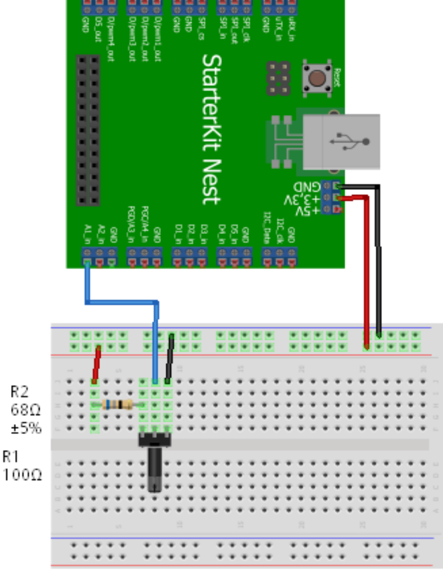
number * minimum step = voltage

For example and Analog Read value of 1000 means:
1000 * 0.002 V = 2V of real voltage input

Note: *Those pins can be used also in digital mode, but are not 5V tolerant! Avoid to connect them to a voltage > 3.3V or you could damage the microcontroller!*

QUESTION: How can I test this feature?
Here is a simple connection of a potentiometer to test the analog input 1:





As you can see from schematics, there is a resistor of 680 Ohm and a potentiometer of 1KOhm. This configuration is made to reduce the max voltage of the A1_in pin. In fact, when the pot is at his max value, at the analog pin we have:

Va1_in = Vdd * (R1 / R1+R2) = 3.3 * (1000 * 1680) = 1.96V

this value is high enough to use almost all the voltage headroom of the analog inputs.
For more info about the using of resistors to reduce a voltage value, see the link
http://en.wikipedia.org/wiki/Voltage_divider

Analog inputs functions

The Flyport APIs have some macros to use and set the Analog inputs.
Initially all the Ax_in pins are just usable as analog inputs, because there is a macro called "**ADCInit()**" that is executed during the initialization of the Flyport.
So, the only thing you have to do to use the analog input functionality is to read the value!

To read the state of a Digital Input → `ADCVal(channel);`

This function returns a int value (but limited to 1023...), and needs the number of the channel (1,2,3 or 4) that are the same of the Flyport pinout.

For example:

```
int myADCValue;   //Initialize the variable to get the value

myADCValue = ADCVal(1); //Insert the value of the
                        //A1_in pin in the myADCValue variable
```

PWMs

Flyport has 4 PWM output pins in standard settings (d1_out, d2_out, d3_out, d4_out), but it can handle up to 9 PWM, using remappable pins (for example digital inputs, or SPI port...).
Pay attention that I2C pins and d5_out cannot be used as remappable pin, so neither as PWMs!
Also the d2_in can't be used as PWM, because it can't handle remappable output modules.

QUESTION: What does PWM mean?
PWM, Pulse Width Modulation, is a digital periodical signal. It is like a square wave, but the duty cycle is modulated. The duty cycle is the ratio between the high level period duration and the low level period duration, often expressed in %.
A duty cycle of 100% is a signal that is always high, a duty cycle of 0% is always low, and 50% is a perfect square wave where the high and low duration are the same.
There are 2 main parameters of a PWM signal: the just known duty cycle, and the frequency of the signal, which represents the repetitions per second of our signal.

For example, a PWM with a frequency of 200Hz will have the period:

$$T = 1/f = 1/200 = 5ms$$

So every 5ms the period will be repeated. Using a PWM with 25% of duty there will be:

- *Total period: 5ms*
- *High duration: (5ms*25/100) = 1.25 ms*
- *Low duration: 5ms – 1.25ms = 3.75 ms*

QUESTION: How can I use PWM signals?
PWMs can be used to create digital modulated signals, but a well known usage is the analog signal creation. Pay attention, it is not a DAC (digital to analog converter), but a simple recreation of a signal with some R-C filters. It is a relative complex operation that depends on the frequency of the PWM and on the load at the pin. For reference, we can assume that we can reproduce good analog signals if them variation (and them higher frequency) is small respect the pwm frequency.

Using PWM with Flyport

PWMs functions of Flyport are basically 4, to give a friendly usage of this feature.

To Initialize the pin → `PWMInit(BYTE pwm, float freq, float dutyc);`

It is the mandatory function that we have to call to setup the a PWM module .

Parameters:
pwm: pwm module number (from 1 up to 9)
freq: the frequency of the signal in Hertz
duty: new duty cycle desired (from 0 up to 100, it is expressed in percentage)

To Turn On the pwm → `PWMOn(Byte io, BYTE pwm);`

It turns on a pwm module and assign it to a pin.

Parameters:
io: io pin to assign at pwm functionality (p1, p2, p3...)
pwm: pwm module number (from 1 up to 9)

To Change Duty Cycle → `PWMDuty(float duty, Byte pwm);`

It can be used to change the pwm duty cycle “on the fly”.

Parameters:
duty: new duty cycle desired (from 0 up to 100, it is expressed in percentage)
pwm: pwm module number (from 1 up to 9)

To Turn Off the pwm → `PWMOff(BYTE pwm);`

It can be used to turn off a pwm module.

Parameters:
pwm: pwm module number (from 1 up to 9)

A simple application of usage of PWM can be the LED bright change. In fact, changing the duty cycle we change the root mean voltage of the signal, so we can change the voltage of the LED with pwm.

For example:

```
PWMInit(1, 1000, 100);  //Initialize pwm1 to work
                        //at 1000 Hz, 100% of duty (always on)

PWMOn(d4out, 1);        //Turns on pwm1, and set it to d4out

PWMDuty(50, 1);         //Change the duty at 50% (about half bright)

PWMDuty(0,1);           //Change the duty at 0% (off)
```

```
PWMOff(1);
```

A more complex example (to put in taskFlyport.c):

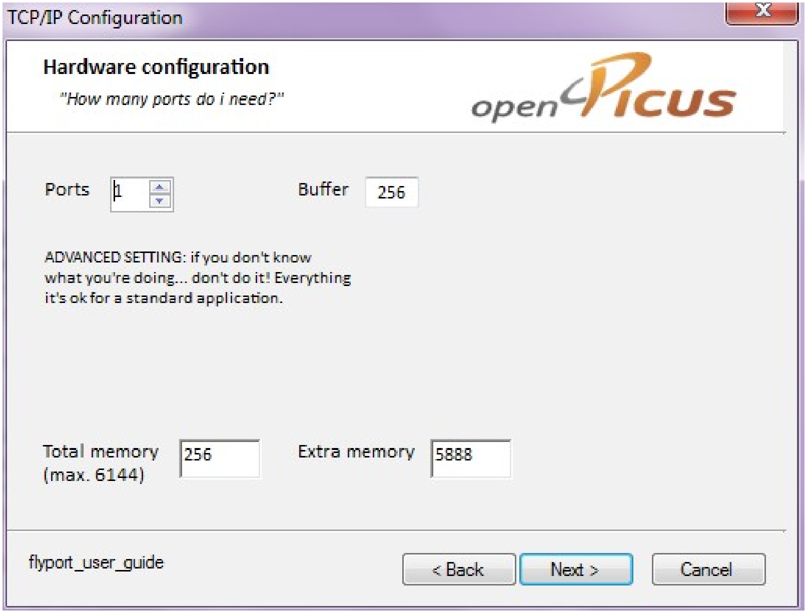
```
#include "taskFlyport.h"

void FlyportTask()
{

const int maxBright = 37;     //here we set max % of brightness
const int minBright = 2;      //and here the min %
float bright = (float)maxBright;

PWMInit(1,1000,maxBright);
PWMOn(d4out, 1);

while(1)
{
      for (bright = maxBright; bright > minBright; bright--)
      {
            PWMDuty(bright, 1);
            vTaskDelay(1);    //used to slow down the effect
      }
      for (bright = minBright; bright < maxBright; bright ++)
      {
            PWMDuty(bright, 1);
            vTaskDelay(1);    //used to slow down the effect
      }
}
}
```

This technique can be used also to change the dc-motor speed, using external circuits to prevent damage of the microcontroller. For more information about PWM and its application see:
http://en.wikipedia.org/wiki/Pulse-width_modulation

Serial communication (UART)

The UART is a serial asynchronous communication module. Basically it is composed by only RX and TX pins, and the clock is implicit. For this reason, you have to know always the baudrate of the signal, or you could not understand the messages, or misunderstand them using a wrong timing.
To help the handle of UART incoming messages, the flyport implements a buffer of 256 characters, that stores the incoming chars automatically. With this setting, you don't have to do any polling on UART rx hardware buffer state every time, but it can be always checked how many chars arrived.

This parameter is customizable by the user, under the wizard tool:



Using this tool, user can customize the maximum number of UART enabled by Framework, and them buffer size.

Warning: *the maximum memory used by UART buffers should not be major than 6144. The wizard tool will show the result on memory occupation on-the-fly.*

Since the USB NEST mount a USB to UART converter, you can try the UART functionalities with a PC and a terminal software like “Putty” or “Termite”, or you can also use the OpenPicus IDE Serial Monitor. Please, **do not use hyper terminal** because it does not properly supports the DTR signal, so it is not compatible with USB NEST!
It is necessary to properly set the baudrate. It is the bit/sec of message timing, so the Flyport and PC programs can know the clock and how to interpreter the information exchanged.

UART functions

There is a sequence to respect before using the UART peripheral module of PIC. In fact, the first thing to do is to properly initiate the module at the desired baud rate. After that, it must be enabled (or turned on).

Note: *In the IDE wizard, there is the possibility to use the uart 1 as "TCP debug on UART1". In this case the UART1 module is initialized at 19200 baud and just turned on.*

To Initialize the module → `UARTInit(int port, long int baud);`

This is a mandatory function that initialize the module to work properly.
This function is just called in the Flyport framework initialization, but could be reused to change the baudrate parameter "on the fly".
Parameters:
port: the UART port to initialize. Note: at the moment the Flyport Framework supports just one UART, but the Hardware allows to create up to four UARTs. Others will be added in the next release, however is possible to create them with standard PIC commands.
Baud: the desired baudrate

To Turn On the module → `UARTOn(int port);`

This function turns on the UART module functionalities. It should be called only after a UARTInit calling.
Parameters:
port: the UART port to turn on

To Turn Off the module → `UARTOff(int port);`

This function turns off the UART module functionalities. It should be called before a UARTInit calling.
Parameters:
port: the UART port to turn off

To know **how many char arrived** → `UARTBufferSize(int port);`

This function returns a int number equal to how many chars are arrived and stored insied the UART RX Buffer.
Parameters:
port: the UART port

Returns:
int N: number of characters arrived

To Read from RX buffer →

```
UARTRead(int port, char *towrite,
                         int count);
```

This function reads characters from the UART RX buffer and put them in the char pointer "towrtite".

It also returns the report of the operation
Parameters:
port: the UART port
towrite: the char pointer to fill with the read characters
count: the number of characters to read

Returns:
int N: N > 0, N characters correctly read.
N < 0, N characters read, but buffer overflow detected.

To Write a string → `UARTWrite(int port, char *buffer);`

This function writes the specific string on the UART port.
Parameters:
port: the UART port
buffer the string to write (a NULL terminated char array)

To Write a character → `UARTWriteCh(int port, char chr);`

This function writes the specific character on the UART port.
Parameters:
port: the UART port
chr the character to write

To Blank the rx buffer → `UARTFlush(int port);`

This function flushes the buffer of the specified UART port.
Parameters:
port: the UART port

I2C communication protocol

The I2C protocol is a 2 wire serial communication. Unlike the UART, I2C has a Master-Slave architecture, where the Master starts all the requests of communication. The baudrate is given by the Master, and the most used values are 100kb/s (low speed) and 400kb/s (high speed). Another difference is that I2C bus needs pull-ups resistor, so needs also of dedicated open collector pins. This is why the I2C pins are not usable by other modules and not remappable.
To use the i2c protocol it is suggested to study its phases of communication, because it has a determinated sequence of operations for starting, stopping, write, read and check if the data was transmitted well. For more information see the link:
http://en.wikipedia.org/wiki/I%C2%B2C

I2C functions

To Initialize the module → `I2CInit(BYTE speed);`
This function initialize the i2c at the specified speed of communication.
Parameters:
speed: it can be used HIGH_SPEED for 400K or LOW_SPEED for 100K

To send a Start condition → `I2CStart();`
This function sends a start sequence on the i2c bus.
Parameters:
none

To send a Restart condition → `I2CRestart();`
This function sends a repeated start sequence on the i2c bus.
Parameters:
none

To send a Stop condition → `I2CStop();`
This function sends a stop sequence on the i2c bus.
Parameters:
none

To Write a byte → `I2CWrite(BYTE data);`
This function writes a byte of data .
Parameters:
data: the byte data to be sent
Returns:
0: NACK condition detected
1: ACK condition detected
2: Write Collision detected
3: Master Collision detected

I2C functions usage library example

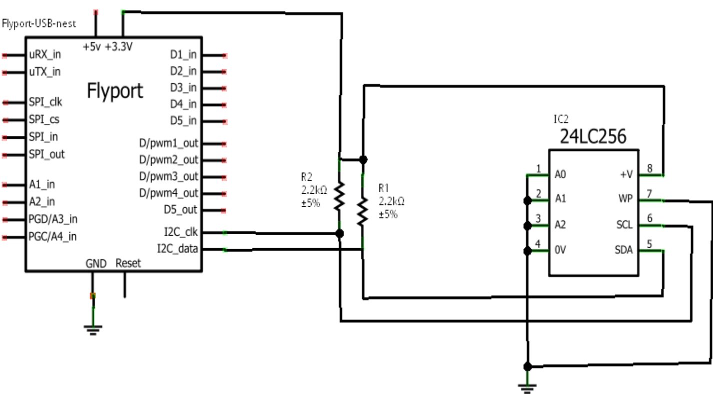
To helps in the complex usage of I2C protocol messages, the library called “Lib I2C EEPROM” implements some Macros to use an external EEPROM. With the help of those macros it can be seen how to use the single functions to write and read a attached I2C slave device.

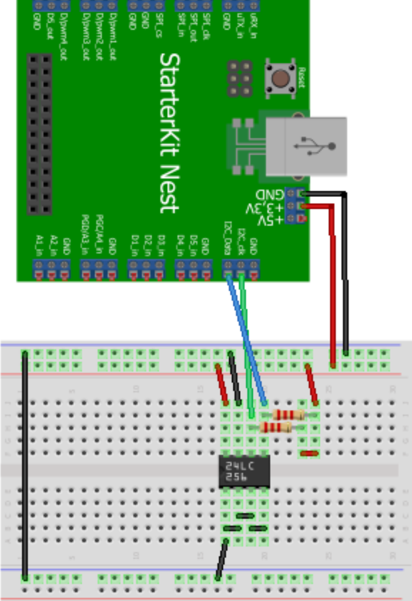
```
#include "taskFlyport.h"
#include "i2ceeprom.h"

void FlyportTask()
{
	EEPROMInit( 0xA0, LOW_SPEED);
	BYTE dataout[298];
	dataout[297]='\0';
	BYTE datain[297]={
			0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x10,
			0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x20,
			0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29, 0x30,
			0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x40,
			0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49, 0x50,
			0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58, 0x59, 0x60,
			0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x70,
			0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79, 0x80,
			0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89, 0x90,
			0x91, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97, 0x98, 0x99,
			0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x10,
			0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x20,
			0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29, 0x30,
			0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x40,
			0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49, 0x50,
			0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58, 0x59, 0x60,
			0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x70,
			0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79, 0x80,
			0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89, 0x90,
			0x91, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97, 0x98, 0x99,
			0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x10,
			0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x20,
			0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29, 0x30,
			0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x40,
			0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49, 0x50,
			0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58, 0x59, 0x60,
			0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x70,
			0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79, 0x80,
			0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89, 0x90,
			0x91, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97, 0x98, 0x99,
		};

	EEPROMWritePage(0x00, datain, 297);

	vTaskDelay(50);

	EEPROMReadBlock(0x00, dataout, 297);

	UARTWrite(1, (char*)dataout);
```

```
        while(1)
        {

        }
}
```

This example is just a short use of the library; for more information please follow the guide inside the "LIB – I2C EEPROM.zip" file in the download section of www.openpicus.com. To use the external helper library, it should be added in the project using the "External Lib" tool of the OpenPicus IDE. To use this tool, used to include files on the "external lib" folder of project, the related files should be outside of project folder to avoid double inclusions of files.

The hardware connections for a 24LCxxx EEPROM is shown below:





Using the TCP/IP stack

Flyport TCP/IP management is based on the Microchip TCP/IP stack. On that basis, the OpenPicus framework integrates the stack with the operating system FreeRTOS, to ease the management of any TCP/IP operation. Even if all the communication issues have been simplified to make everything easier, the TCP/IP is a very complex stack with many functionalities, so a minimum knowledge of the TCP/IP is needed (if you are not able to make the basic configuration of an access point, of a wireless router, maybe you will have some problem in integration of Wi-Fi with your application).

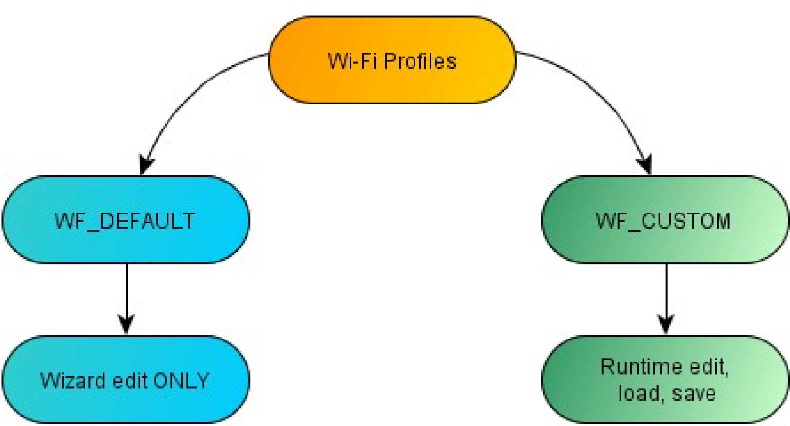
Managing the Wi-Fi

⚠ IMPORTANT: **Flyport cannot work as access point. In ad-hoc mode, the DHCP server of the Flyport assign the IP address only to one device. Ad-hoc networks are peer-to-peer networks. In infrastructure mode, Flyport connects to already existent networks.**

How to manage all the aspects of the Wi-Fi connections, how to monitor the status of the connections and how to modify all the parameters of the Wi-Fi.

The connection profiles

Flyport allows the user to freely set any parameter for the Wi-Fi connection, SSID, encryption, DHCP and so on. All the configuration parameters can be set using the IDE TCP/IP wizard and those configuration will be stored in the "WF_DEFAULT" connection profile.
In the Flyport Framework there are two Wi-Fi Profiles, the default one named WF_DEFAULT and the customizable one named WF_CUSTOM.



QUESTION: What are the profiles?

The profiles are structure were are stored basic Wifi information, necessary to Flyport to know how to connect to the network. The basic informations are:

- IP of the device
- DNS servers
- default GATEWAY
- DHCP enable / disable
- netbios and SSID name
- network type (adhoc or infrastructure)
- Security configuration parameters (type and password)

QUESTION: How can be used the profiles?

The profiles can be used inside the function that starts a Wifi connection. This connection is

```
WFConnect( connection profile);
```

and can be used with “WFConnect(WF_DEFAULT);” or with “WFConnect(WF_CUSTOM);”

QUESTION: How can I customize the profiles?

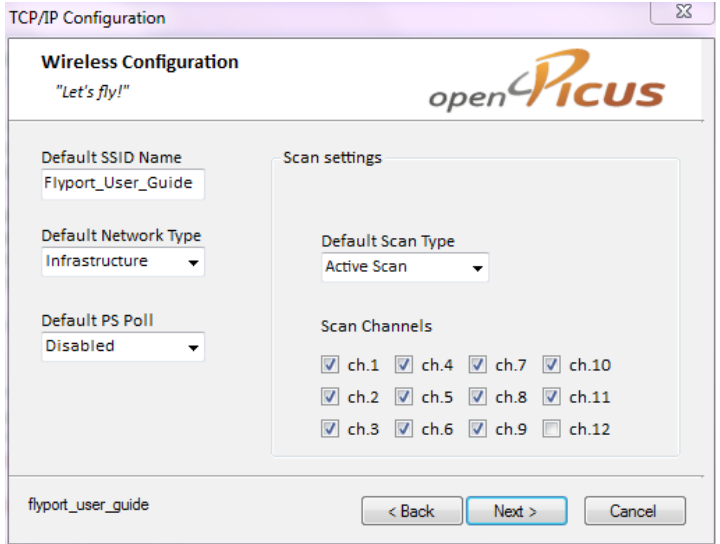
Actually there are two ways to customize a connection profile.

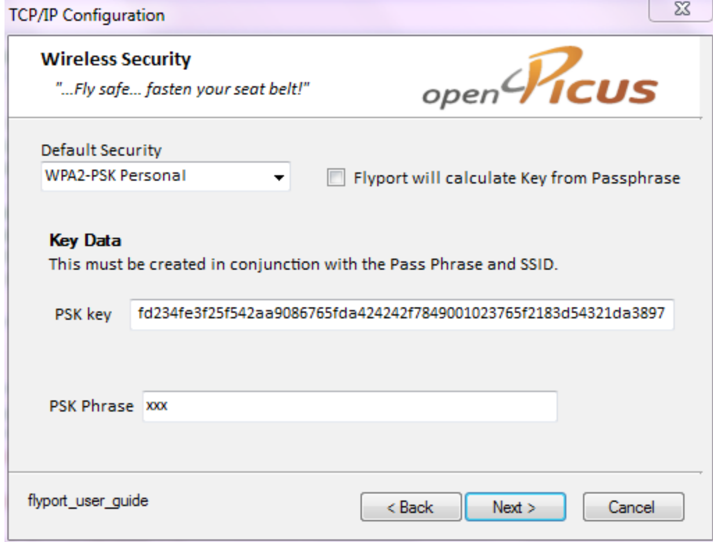
The **WF_DEFAULT** is the standard profile, and this type of connection is used by the framework in the standard template. Its parameters can be changed with the Wizard tool of the IDE. Those values are fixed on PIC memory and can't be changed in runtime. Every time you change some parameters in the wizard you have to compile again the firmware.

The **WF_CUSTOM** is the runtime configurable *profile template*. Its parameters can be updated in the Flyport tasks, but all the changes will be erased after a Flyport reboot! In fact, WF_CUSTOM is the same of WF_DEFAULT at startup, so every change should be reloaded by user application. Every time the Flyport is restarted, the WF_CUSTOM profile should be loaded from memory, and before a restart the values should be saved in memory by user.

Connection functions

To handle the Wi-Fi connections the framework supports some user functions. Some of them are usable to set the connection profile, the others to use those profiles to establish the just setted up connection. To help user with the settings, there is a graphical wizard in OpenPicus IDE, but you could change this parameters with a simple text editor; we suggest you to use the wizard, since it is a easy and fast tool, and it compiles project modification automatically after changes.





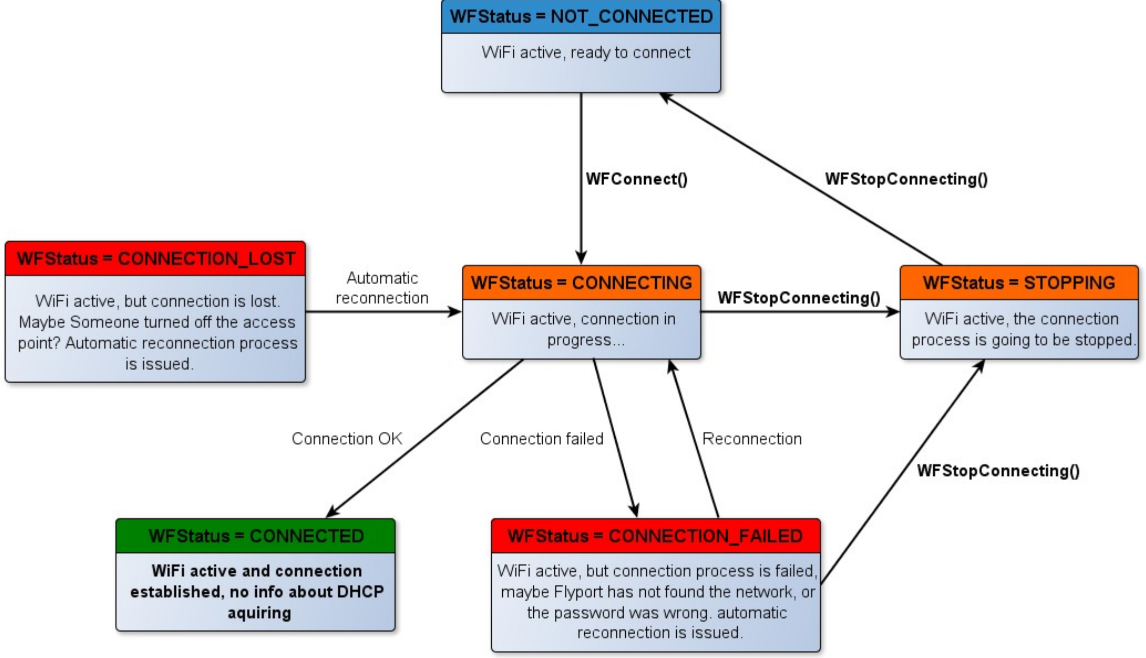
For more info about the IDE Wizard please refer to the OpenPicus IDE Manual.

QUESTION: How can I check Wi-Fi connection?
Actually there is a very important variable called **WFStatus**. Its value is directly dependent on Wi-Fi connection status, and can be (them are defined in Hwlib.h):

- NOT_CONNECTED
- CONNECTING
- CONNECTED
- CONNECTION_LOST
- CONNECTION_FAILED
- STOPPING
- TURNED_OFF

Those values defines different statuses, and different "jobs in progress".

The status "**NOT_CONNECTED**" and "**TURNED_OFF**" are ***static status***, and them can be changed only with a user task input. The "NOT_CONNECTED" is when Wi-Fi connection is closed and there is no need to open it. The "TURNED_OFF" is a status when Wi-Fi module is turned off for power saving. The other status are ***dynamic*** and can change without user input, but by *external events*. In fact when user try to connect to a Wi-Fi network the WFStatus passes from "NOT_CONNECTED" to "**CONNECTING**". In this status show that Wi-Fi module is trying to connect to a network. If it connects, WFStatus passes to "**CONNECTED**", if it fail it passes to "**CONNECTION_FAILED**". If this last status occurs OpenPicus Framework automatically retries to connect to Wi-Fi network until a success is reached or user stops the connection from user task. If a connection is lost for network problems (for example no reply from router, or too much distance between Flyport and other Wi-Fi device) the WFStatus become "**CONNECTION_LOST**" and framework automatically retries to connect to network. Those combination of values of WFStatus variable can be used in user task to control the Wi-Fi connection, and try for example different parameters, or to stop the retries on a limited number for a certain time.



To Connect to a Wi-Fi net → `WFConnect( int pconn );`

This function starts the opening of a connection at the provided WiFi profile. The Flyport will continue its tasks even if it can't connect to the network, and will send UART messages for connection time-out
Parameters:
pconn: the Connection Wi-Fi Profile (WF_DEFAULT or WF_CUSTOM)

To Disconnect from a Wi-Fi net → `WFDisconnect();`

This function starts the closing of the actual network
Parameters:
none

To Stop Connection retries to a Wi-Fi net → `WFStopConnecting( );`

When the command WFConnect is launched, the device tries to connect to the selected Wi-Fi network until it doesn't find it. If it is needed to STOP the retries, this function must be called
Parameters:
pconn: the Connection Wi-Fi Profile (WF_DEFAULT or WF_CUSTOM)

QUESTION: How can I know what kind of connection profile I have used?
The WFConnection variable stores the Wi-Fi profile number used to connect Flyport to Wi-Fi network. In fact it can be used the “if(WFConnection == WF_CUSTOM)” to know if a Wi-Fi custom profile is used or not.

Customizing network parameters at runtime

To Customize the WF_CUSTOM profile the framework supports some user functions. Some of them are for set parameter, other for save/load/check profiles.

To Set a parameter of WF_CUSTOM → **`WFSetParam(int paramtoset, char * paramstring );`**

Parameters:

paramtoset: the parameter of profile to change.

- MY_IP_ADDR
- PRIMARY_DNS
- SECONDARY_DNS
- MY_GATEWAY
- SUBNET_MASK
- NETIOS_NAME
- SSID_NAME
- DHCP_ENABLE (ENABLED or DISABLED)
- NETWORK_TYPE (ADHOC or INFRASTRUCTURE)

paramstring: the string value of parameter.

To Set the Security Parameters of WFCustom → **`WFSetSecurity(BYTE mode, char* keypass, BYTE keylen, BYTE keyind);`**

Parameters:

mode: the security mode. Following are valid parameters:

- **WF_SECURITY_OPEN** : no security
- **WF_SECURITY_WEP_40**: WEP security, with 40 bit key
- **WF_SECURITY_WEP_104**: WEP security , with 104 bit key
- **WF_SECURITY_WPA_WITH_KEY**: WPA-PSK personal security, the user specifies the hex key
- **WF_SECURITY_WPA_WITH_PASS_PHRASE**: WPA-PSK personal security, the user specifies only the passphrase
- **WF_SECURITY_WPA2_WITH_KEY**: WAP2-PSK personal security, the user specifies the hex key
- **WF_SECURITY_WPA2_WITH_PASS_PHRASE**: WPA2-PSK personal security, the user specifies only the passphrase
- **WF_SECURITY_WPA_AUTO_WITH_KEY**: WPA-PSK personal or WPA2-PSK personal (the Flyport will auto select the mode) with hex key
- **WF_SECURITY_WPA_AUTO_WITH_PASS_PHRASE:** WPA-PSK personal or WPA2-PSK personal (autoselect) with pass phrase

keypass: the key or passphrase fot the network A key must be specified also for open connections (you can put a blank string, like “”).

keylenght:lenght of the key/passphrase. Must be specified also for open connections (can be 0).

keyind: index of the key (used for WEP security, but must be specified also for all others, in that case can be 0).

NOTE: *For WPA/WPA2 with passphrase, the Flyport must calculate the hex key. The calculation is long and difficult, so it will take about 20 seconds to connect!*

To Save parameter of WF_CUSTOM → **`WFCustomSave( );`**

To prevent the loosing of all the data of custom profiles (that are normally stored in RAM), this function store all values in internal non volatile Flash memory. The saved parameters should be loaded after every reboot of Flyport to be used.

To Load parameter of WF_CUSTOM → **`WFCustomLoad( );`**

Load the previously saved parameters of WF_CUSTOM.

To Detect existing saved parameter of WF_CUSTOM → **`WFCustomExist( );`**

Verifies if in memory is present some data for the WF_CUSTOM profile. It can be useful at the startup of the device to check if in a previous session (before any power off) has been saved some configuration data, so firmware can choose if Custom Parameter should be loaded from memory, or if in previous session there were not saved any WFCustom parameters.

To Delete existing saved parameter of WF_CUSTOM → **`WFCustomDelete( );`**

Wi-Fi functions and variables

There are also some more functions and system variables to control the Wi-Fi state, and use the Hibernation mode:

APP_CONFIG **AppConfig** is a variable which contains a lot of Wi-Fi parameters. It is a APP_CONFIG struct type. The majority of parameters are just available in WF_DEFAULT and in WF_CUSTOM, but the IP address of Flyport can change from its default value if DHCP is enabled.
To access to this specific parameter, it can be used the notation **AppConfig.MyIPAddr**. This will return the *IP_ADDR* variable that store the effective value of Flyport's IP address, even if it was changed by other devices.

To use the AppConfig.MyIPAddr values as four different bytes:

- *AppConfig.MyIPAddr.byte.LB*
- *AppConfig.MyIPAddr.byte.HB*
- *AppConfig.MyIPAddr.byte.UB*
- *AppConfig.MyIPAddr.byte.MB*

In “Helpers.c” there is also a function to convert a generic string to *IP_ADDR* variable.
Its statement is:

```
BOOL StringToIPAddress( BYTE* str, IP_ADDR* IPAddress);
```

This function returns TRUE if the string provided were converted to IP Address, or FALSE if the process was concluded unsuccessfully.

To use the AppConfig.MyIPAddr byte values as a single string format, them can be converted using the helper function

```
void IPAddressToString(IP_ADDR* IPAddress, char* ipString);
```

tWFNetwork is a structure containing all the network parameters:

- BYTE **bssid** [WF_BSSID_LENGTH]
- CHAR **ssid** [WF_MAX_SSID_LENGTH+1]
- UINT8 **channel**
- UINT8 **signal**
- BYTE **security**
- BYTE **type**
- UINT8 **beacon**
- UINT8 **preamble**

WFScan() is a utility function to start the scan of all the available Wi-Fi networks.
All the scanned functions will be indexed with a int number.
Note: *this function can't be called inside “WF_Event.c”!*

WFNetworkFound() return a *int* number of how many networks are found. At startup it returns 0. This function is useful to know the maximum index of the networks found, 0 for no Wi-Fi networks found yet.

WFScanList(int n) reads the parameter of the discovered network number **n** and returns a *tWFNetwork* variable, with all the parameters read.

The use of this function could be:

```
tWFNetwork NetData;
int indexNetwork = 1;
NetData = WFScanList( indexNetwork );
UARTWrite(1, NetData.ssid);
```

NOTE: *for more informations on the using of Network Scanner Functions, please refer to "APP – NetworkScanner" in the download section of www.openpicus.com*

TCP Protocol

TCP, Transmission Control Protocol, is a packet oriented protocol. It can be used to transmit and receive packets of data through the network between server and client.

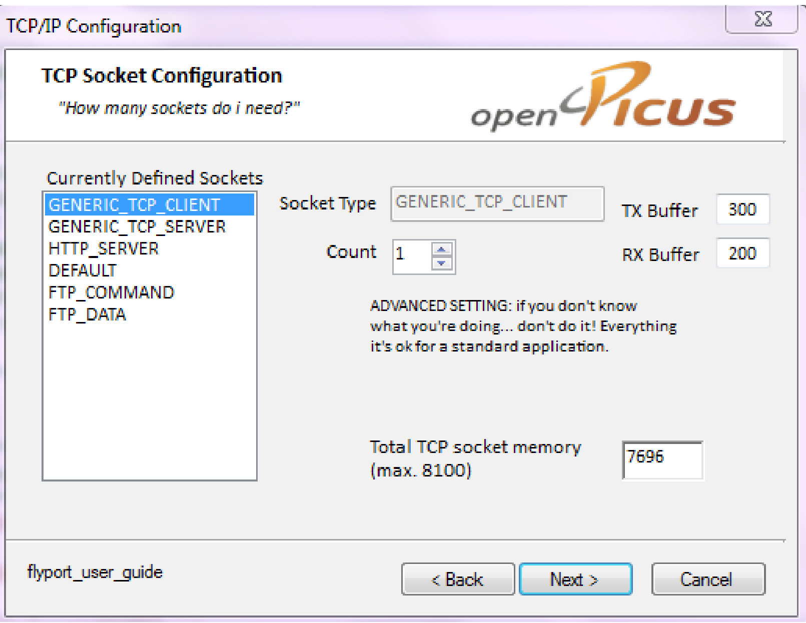
- To use this kind of transmission, server and client should estabilish a connection. Since the connection between server and client consumes network resources (exchanging protocol service packets), it should be closed after the transmission is finished.
- In every TCP connection there is a Server and a Client, and Flyport can play both roles.
- The TCP/IP transmission is “full-duplex” capable, and all the packets sended arrive at source ordered and “at most once”.
- With the using of acknowledgement, timeouts and checksums, the TCP protocol can check the integrity of every packet and know if a packet arrived at destination.
- At every host, multiple connections can be opened with the using of different ports. We will call the “TCP_SOCKETS”.

TCP Functions

Every TCP connection is identified by its *TCP_SOCKET.*

First of all *it must be created our TCP_SOCKETs:*
TCP_SOCKET **myTCPSocket** = INVALID_SOCKET;

Every TCP_SOCKET needed by application must be properly sized in IDE Wizard, to optimize Flyport memory. It should be less possible both number of sockets and buffer length.



The parameters to set are the same for “GENERIC_TCP_CLIENT” and for “GENERIC_TCP_SERVER”.

Them are:

- **Count:** the number of Socket for connection to use; them can be different from client and server
- **TX Buffer:** the maximum dimension of transmission buffer
- **RX Buffer:** the maximum dimension of receive buffer

The total TCP socket memory is the maximum space that can be used for TCP sockets inside the Wi-Fi module's memory. The TCP buffers are not stored in microcontroller's memory but in Wi-Fi module's one, so the using of TCP sockets does not affect PIC's memory occupation. Anyway a max of 8100 bytes can be used, and them are the sum of all TCP sockets that are listed in the left list.

The "GENERIC_TCP_XXXXXX" TX and RX Buffer dimension are the MAXIMUM dimension of data that can be transmitted or received in a single packet. If the user application needs to send 500 bytes of packets length in TCP server connection, the relative TX Buffer must necessary be bigger, the same for RX Buffer.

The TCP_SOCKET status is *invalid* when the TCP connection is not working. Checking the status of our TCP socket is a way to know if the connection is working and well opened.

There are two connection types, one where the Flyport is Server and one where the Flyport is Client.
If *Flyport is Server* the parameter needed is only the port number;
if *Flyport is Client* the parameters needed are port number and IP address of Server.

To open a TCP connection → myTCPSocket=TCPServerOpen(char[] tcpport);
or
→ myTCPSocket=TCPClientOpen(char[] tcpaddr, char[] tcpport);

To close a TCP connection → TCPServerClose(myTCPSocket);
or
→ TCPClientClose(myTCPSocket);

It can be useful to detach a client when Flyport has a server role. This is not a connection close, but maintain it opened and free only the connection for another client.

To detach a client → TCPServerDetach(myTCPSocket);
It disconnect a Client from connection and pass to a listening status. Another client could connect to Server in this manner

To check a TCP connection → TCPisConn(myTCPSocket);
Returns:
connection state: true TCP is connected
false TCP is NOT connected
This function is useful to check if a TCP connection is available

The data exchange can be full-duplex, so both Server and Client can send and receive packets.

To write data → **TCPWrite(TCP_SOCKET socktowrite, char* writech, int wlen);**
Parameters:
socktowrite: the tcp socket connection to write
writech: the char array of data to write out
wlen: the length of data to write

NOTE: *The receive method is a little bit different. In fact it should be checked before how many chars arrived from TCP connection, and after call the read function.*

To know the rx length → **TCPRxLen(TCP_SOCKET socklen);**
Parameters:
socklen: *the TCP socket to check*
Returns:
WORD number of bytes available to be read

To read the rx buffer → **TCPRead(TCP_SOCKET socktoread, char[] readch, int rlen);**
Parameters:
socktoread: *the TCP socket to read*
readch: *the char array to fill with the read characters*
rlen: *the number of characters to read (the word returned by TCPRxLen);*

Warning
*The length of the array must be **AT LEAST = rlen + 1**, because at the end of the operation the array is automatically NULL terminated (is added the '\0' character).*

TCP usage

Basic usage example of TCP Client and Server connection
In this short example of how user should configure the taskFlyport.c file to use a TCP connection in both Server and Client mode with the usage of two different sockets.

```
#include "taskFlyport.h"

void FlyportTask()
{
	TCP_SOCKET SocketTCPServer = INVALID_SOCKET;
	TCP_SOCKET SocketTCPClient = INVALID_SOCKET;

	BOOL clconn = FALSE;
	BOOL clconnClient = FALSE;

	WFConnect(WF_DEFAULT);
	while (WFStatus != CONNECTED);
	UARTWrite(1,"Flyport connected... hello world!\r\n");

	const char txstring[6] = "hello!";
	const int txstringlen = 6;
	int tcprxlength = 0;
```

```
IOPut(d5out, off);

while(1)
{
        // Check TCP Server Connection Activity
        if(TCPisConn(SocketTCPServer))
        {
                if (clconn == FALSE)
                {
                        clconn = TRUE;
                        IOPut(D4Out,on);
                }
        }
        else
        {
                if (clconn == TRUE)
                {
                        clconn = FALSE;
                        IOPut(D4Out,off);
                }
        }

        // Check TCP Client Connection Activity
        if(TCPisConn(SocketTCPClient))
        {
                if (clconnClient == FALSE)
                {
                        clconnClient = TRUE;
                        IOPut(D5Out,on);
                }
        }
        else
        {
                if (clconnClient == TRUE)
                {
                        clconnClient = FALSE;
                        IOPut(D5Out,off);
                        TCPClientClose(SocketTCPClient);
                        SocketTCPClient = INVALID_SOCKET;
                }
                else
                {
                       TCPClientClose(SocketTCPClient);
                        SocketTCPClient = INVALID_SOCKET;
                }
        }


      //Create socket Server connection
      if(SocketTCPServer == INVALID_SOCKET)
      {
            SocketTCPServer = TCPServerOpen("2050");
      }
      else
      {
            tcprxlength = TCPRxLen(SocketTCPServer);
```

```
                        if(tcprxlength > 0)
                        {
                                TCPWrite(SocketTCPServer, tcprxlength, 1);
                        }
                }

                //Create socket Client connection
                if(SocketTCPClient == INVALID_SOCKET)
                {
                        SocketTCPClient = TCPClientOpen("192.168.1.107","2051");
                        vTaskDelay(50);
                }
                else
                {
                        tcprxlength = TCPRxLen(SocketTCPClient);
                        if(tcprxlength > 0)
                        {
                                TCPWrite(SocketTCPClient, tcprxlength, 1);
                        }
                }
        }
}
```

UDP Protocol

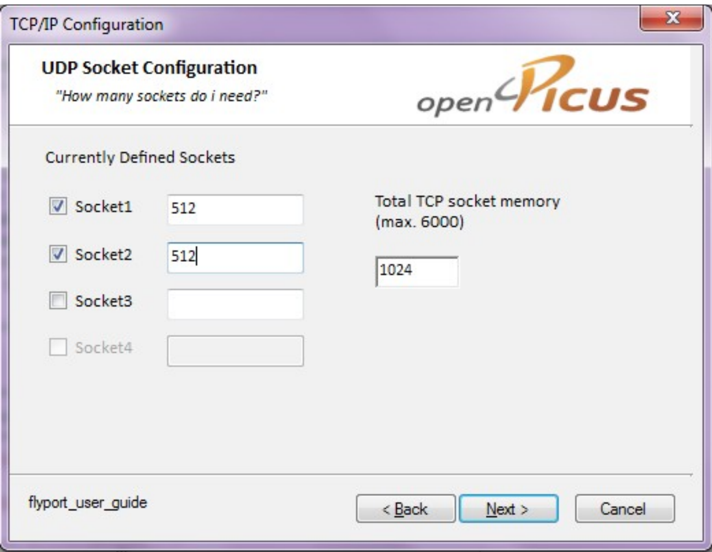
The UDP (User Data Protocol) is another packet oriented protocol. This kind of protocol is very similar to TCP, but it has no flow controlling system. In fact, the UDP header bytes are less than TCP.

- UDP is a connectionless protocol
- Packets can't be reordered and recovered with acknowledgement checks
- Faster than TCP
- Checksum error detection
- Can perform a Server to Multi-Client transmission

QUESTION: What does it mean "UDP is connectionless"?
UDP is *connectionless* in the meaning that Server does not worry about the Clients, and it sends information even if no clients are connected. In TCP it must be present a connection between server and client, in UDP this kind of connection could be not required. A simple example is Internet Radio Service, where the stream of data is always available, even if no clients are present.
With a Server to Multi-Client connection the stream of data is one way: server can only transmit, and can't receive with the same UDP Socket.

UDP Functions

There are up to 4 UDP SOCKETs configurable in Wizard. Every Socket has its own dimension, and a total of 6000 bytes is the maximum space available for all of them.
Every UDP Socket is represented by its number. The *sock* variable used in UDP Functions is basically a BYTE that stores the number of the UDP Socket used when the connection is opened.



In the IDE wizard there is the above configuration page, where there can be setted the UDP sockets.

QUESTION: How can I open a UDP connection?
There are 3 ways to open a UDP Connection:
Flyport can be as **Server** for point to multipoint (*broadcast*, only transmission is available), or point to point (*server*, both tx and rx available). In this case the framework needs only the port number to open for the connection.
Flyport can also open a connection as **Client**, but in this case framework needs the Server IP Address and also the udp port.

To Open Broadcast connection → UDPBroadcastOpen(char[] udpport);
This function opens a server broadcast connection at the specific port number
Parameters:
udpport: the port to open for the connection
Returns:
BYTE of Socket number

To Open Server connection → UDPServerOpen(char[] udpport);
This function opens a server point to point connection at the specific port number
Parameters:
udpport: the port to open for the connection
Returns:
BYTE of Socket number

To Close them (both are server) → UDPServerClose(BYTE sock);
This function close the server connection at the specific socket number
Parameters:
sock: BYTE of Socket number

To Open Client connection → UDPClientOpen(char * udpaddr, char[] udpport);
This function opens a client connection at the specific server address and port number
Parameters:
udpaddr: the server ip address
udpport: the port to open for the connection
Returns:
BYTE of Socket number

To Close Client connection → UDPClientClose(BYTE sock);
This function close the client connection at the specific socket number
Parameters:
sock: BYTE of Socket number

Every UDP Socket has its own RX buffer, so the reading is handled automatically by Operating System.

To know the RX bytes length → UDPRxLEn(BYTE sock);
This function reads the number of bytes available at the specific socket number
Parameters:
sock: BYTE of Socket number
Returns:
WORD of available bytes at RX Buffer

To Read them → **UDPRead(BYTE sock, char str2rd[], int lstr);**
This function reads the RX buffer and puts them at str2rd from the specific socket number
Parameters:
sock: BYTE of Socket number
str2rd: the char array to copy the Rx buffer content
lstr: the length of string to read
Returns:
INT of bytes read from RX Buffer

To Write a packet → **UDPWrite(BYTE sock, BYTE *str2wr, int lstr);**
This function write the str2wr char array to the specified socket number
Parameters:
sock: BYTE of Socket number
str2wr: the char array to write to TX buffer
lstr: the length of string to write
Returns:
WORD of bytes wrote to TX Buffer

UDP usage example

Here is a basic example of usage of UDP protocol in with Flyport as Server. As it can be seen on example UDP is full-duplex, but the connection with client is not necessary, and UDP server can write strings also when no clients are connected

```
#include "taskFlyport.h"

int serverUDPsocket;
int serverRxLength = 0;
char serverString [512];

void FlyportTask()
{
	int i = 0;

	WFConnect(WF_DEFAULT);
	while (WFStatus != CONNECTED);
	UARTWrite(1,"Flyport connected... hello world!\r\n");

	// Open Server UDP connection
	serverUDPsocket = UDPServerOpen("5010");

	while(1)
	{
		// wait 0.5 sec
		vTaskDelay(50);

		if(!serverUDPsocket)
		{
			UARTWrite(1, "unable to open server UDP socket\r\n");
		}
		else
		{
			// write a string via UDP!
			UDPWrite(serverUDPsocket, "Hello!\r\n", 6);

			// Check Server RX length
			serverRxLength = UDPRxLen(serverUDPsocket);

			// Check if server received some datas
			if(serverRxLength > 0)
			{
				UDPRead(serverUDPsocket, serverString, serverRxLength);
				UARTWrite(1, serverString);
				UARTWrite(1,"\r\n");
				// Clear serverString buffer for the next using
				for (i=0; i<serverRxLength; i++)
					serverString[i] = 0;
			}
		}
	}
}
```

SMTP Protocol

The SMTP (Simple Mail Transfer Protocol) is used to send emails, and Flyport can send emails using a No-SSL connection. User can customize at runtime sender, receiver, subject and message.

The usage of SMTP is not recommended for frequently communication, since it can overflow servers. For example it should be used for events of occasional errors, or daily reports or similar tasks.

QUESTION: How can I use SSL connections?
SSL is not free of charge on Microchip's TCPIP stack, and needs to be brought by user.
At this time we don't provide SSL implementation on OpenPicus Framework.

Here is a simple example on how to use SMTP feature with a mail-server that don't needs SSL connection:

```
/**********************************************
**    SMTP report test
**********************************************/

#include "taskFlyport.h"

void FlyportTask()
{
      char reportResult[30];

      WFConnect(WF_DEFAULT);
      while (WFStatus != CONNECTED);
      UARTWrite(1,"Flyport connected... hello world!\r\n");
      vTaskDelay(200); //very important if you use DHCP client

      // wait for the SMTP Start
      while(!SMTPStart());
      UARTWrite(1,"SMTP Started!\r\n");
      SMTPSetServer(SERVER_NAME, "server.com");
      SMTPSetServer(SERVER_USER, "username");
      SMTPSetServer(SERVER_PASS, "password");
      SMTPSetServer(SERVER_PORT, "25");
      SMTPSetMsg(MSG_TO, "destination@servermail.com");
      SMTPSetMsg(MSG_BODY, "Flyport SMTP!" );
      SMTPSetMsg(MSG_FROM, "sender@server.com");
      SMTPSetMsg(MSG_SUBJECT, "Flyport SMTP test!!");
      UARTWrite(1, "Client initialized!\r\n");
      vTaskDelay(100);

      // wait for sending complete
      if(SMTPSend())
      {
            UARTWrite(1,"sending email...\r\n");
            while(SMTPBusy() == TRUE)
            {
                  UARTWrite(1, ".");
                  vTaskDelay(10);
```

```
            }
            UARTWrite(1,"Email sent!\r\n");
        }
        else
        {
            UARTWrite(1,"Error in email sending\r\n");
        }

        WORD report = SMTPReport();
        sprintf(reportResult, "report: %d\r\n", res);
        UARTWrite(1, reportResult);

        while(1)
        {

        }
}
```

QUESTION: What are the values of SMTPReport?

The SMTP reports values could be:

- SMTP_SUCCESS (0x0000) , email sended
- SMTP_RESOLVE_ERROR (0x8000) , server not found (unresolved DNS)
- SMTP_CONNECT_ERROR (0x8001) , unsuccessful connection (no reply from server)

Other values can be handled, and are standard values used in every email protocol

FTP Client

FTP Client can handle the file exchange between Flyport and PC. Anyway big files can't be handled due limited Flyport's memory resources.

Flyport can only be Client, so other devices have to provide the FTP server service (for example a PC with FileZilla Server....)

Basically, a FTP connection is a special set of instructions through TCP protocol. With those commands it can be controlled the file and directories handle, user login and logout, etc...
To use FTP Client connection are needed two different sockets: FTP_COMMAND and FTP_DATA.

Those sockets are used to exchange datas in passive mode operation. The passive mode operation is basically a TCP connection at a defined port (reserved to FTP) with some specific command.
To use this feature, please refer to FTP passive mode guides, and use the FTP functions like the similar TCP functions.

The webserver and HTTPApp.c

The **webserver** is a great feature of Flyport. It permits to control and access to Flyport's functionalities with the using of a simple *web browser*, could it be installed on a PC, a smartphone or a tablet. The whole work behind webserver service is made by *HTTPApp.c* and webserver files, with the using of dynamic variables. The webserver task is indipendent from user task, so the two things could be seen as separated systems (like two different softwares on a PC, running together on the same machine) that interacts together with some specifics functions.

What is a webserver and how it works

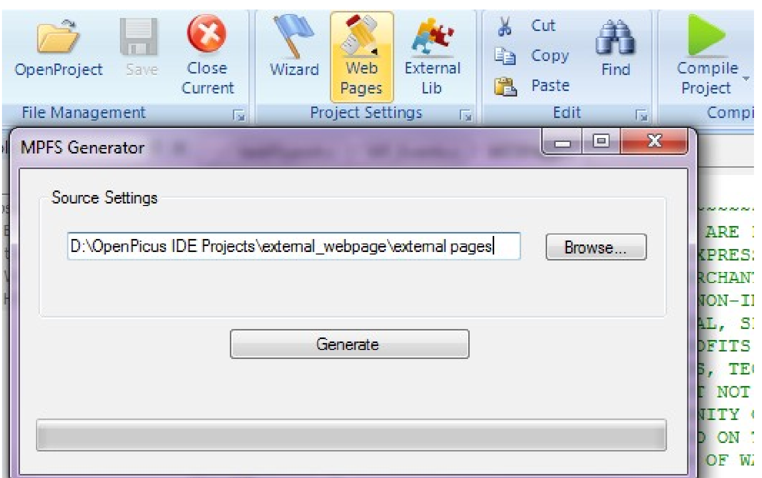
A **webserver** is a html page that browser interpretes and shows with text, graphics and interactive contents. The final result is generally a combination of html code and multimedia files, using also “smart” scripts.
The access of content between webserver and a client (the web browser) is a file exchange. All the HTML pages, images, sounds and scripts are files that client downloads from server and use to show up the web page. To download those files, client requests them at server that handles the data exchanges. Once the client has enough informations, it renders the page in the way we use to see always on PC screens and tablet or smartphone displays.

Flyport webserver and how it works

Flyport webserver is not different than a conventional webserver. Flyport gets the role of server if a browser tries to render its webpage and sends to clients its webserver files. Files can be images, htm, scripts and so on.

The only difference between Flyport and conventional webservers is in the hardware structure. A embedded device is not a professional server, it remains a embedded device and its memory, its bandwidth and its calculation power are limited... don't try to stream 2hours of HD video with a 16MIPS microcontroller please, it could be a bad experience!
Equally as other webservers, Flyport too stores files in its memory as “index.htm” page or “Openpicus.jpg” image or “status.xml” or “mchp.js” for example, and those files should be as smaal as possible due to limitated internal memory of microcontroller itself.

QUESTION: How can I create a webserver page?
Every HTML page can be written with any simple text writing tools, for example Notepad or Notepad++. There are lots of dedicated software for html code writing, and them could be used all due to the using of importing functionalities.
The OpenPicus IDE helps developers on the tedious import phase with a simple helper tool (derived from Microchip's MPFS file system generator), called “Web pages”. It permits user to select a folder to import; after pressing the button “Generate” the webserver inside the folder will be automatically converted and compiled to be used with OpenPicus Framework and Flyport module.



NOTE: *You should name your html page as "index.htm" to let it be shown as the main webserver page. If you would like to have a multipage structure, be sure that the very first page is named "index.htm" or your browser could display the* ***404 page not found error.***

Dynamic variables

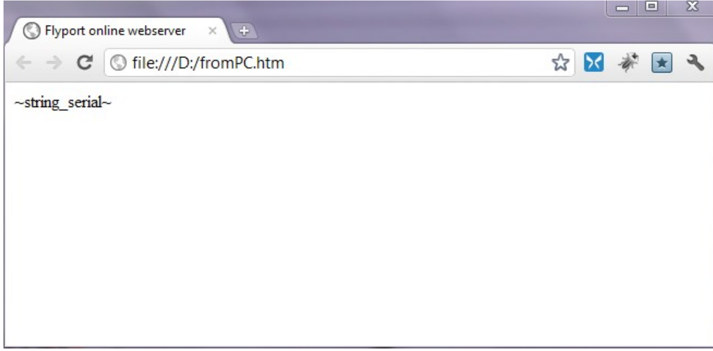
A dynamic variable is a variable that could be changed by OpenPicus Framework at runtime. This mean that its value is not pre-defined but can be changed, for example by a Serial command or the state of a input pin. This is one useful feature of webserver, since it permits to check data changes by the view of a webpage, and not only with serial communication, LCD display or other physical debugging tools... Using webserver, Flyport can also draw graphics on the state of its peripheral status, or its memory value or anything user should need.

A dynamic variable is a variable with the `~this_is_a_dynamic_var~` statement, were the two "~" informs OpenPicus Framework that the value of the variable is connected to a variable content, like a number or a string, and that it should load it with the using of specific functions inside of *HTTPApp.c* file.

QUESTION: How can I use dynamic variables?
Here is an example of the use of dynamic variables. We have a almost blank webpage, were the only text shown is a dynamic variable. The code of the *index.htm* page is this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<title>Flyport online webserver</title>
</head>

<body>
~string_serial~

</body>
</html>
```

As it can be seen, the only content of the body of this webpage is the `~string_serial~` dynamic variable. If this index.htm is opened with a browser in a pc, the result will be like this:



In this case, since there is no Flyport to change the dynamic variable value, the browser takes the statement as a constant text.

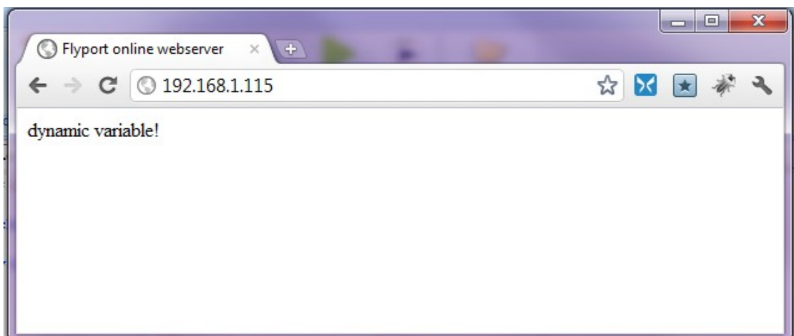
What happens in Flyport's webserver? If we define the variable *string_serial* inside our "*taskFlyport.c*" as:

```
char string_serial[20] = "dynamic variable!";
```

the webserver, when should return the value of `~string_serial~` will return the VALUE of our char array called `string_serial` using the related callback function placed by user in *HTTPApp.c*:

```
HTTPPrint_string_serial()
{
	TCPPutString(sktHTTP,(BYTE *)string_serial);
}
```

This specific callback function executes a TCP string write to our sktHTTP that is the current HTTP-socket.
Those callback functions are named automatically with HTTPPrint_*variablename* name by the webpage import tool, by the using of `~variablename~` statement, for every dynamic variable founded.
The webpage import tool puts the declaration of every "*void HTTPPrint_variablename();*" function inside the file **HTTPPrint.h**, so OpenPicus IDE will generate a compiler error if all the callback functions are not written inside HTTPApp.c by the user.
This will be the result on a web browser that loads the Flyport's webserver page:



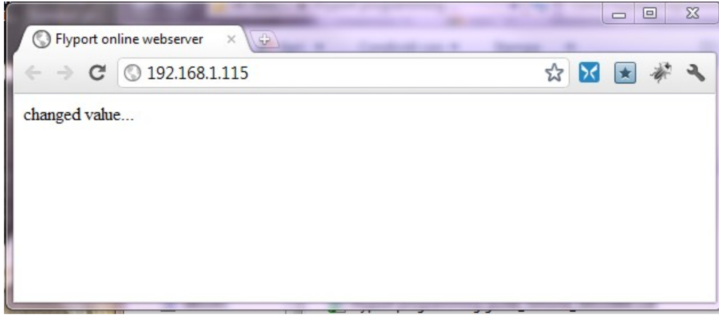
NOTE: *The address pointed by web browser (http://192.168.1.115/) is the IP address of Flyport in the network. This is a way to access to the webserver, as Flyport is located on a local network.*

QUESTION: How can I change value of a dynamic variable?
The value "dynamic value!" is the value that Flyport take from the variable *~string_serial~* when the page is given to a client due a request of access to the page. If the value of the variable changes its status, the webpage will not be changed since there is no refresh mechanism inside the code of this specific *index.htm. Here's* another example of what appens when a dynamic variable is changed at runtime.

Here is the code of *"taskFlyport.c"* that permits user *to change* the string *"string_serial"* by runtime execution of User Firmware, with the using of UART1:

```
#include "taskFlyport.h"

char string_serial[20]="dynamic variable!";

void FlyportTask()
{
	WFConnect(WF_DEFAULT);
	while (WFStatus != CONNECTED);
	UARTWrite(1,"Flyport connected... hello world!\r\n");
	// Write to serial console the status of string_serial
	UARTWrite(1,string_serial);
	int len_s;

	while(1)
	{
		// wait until a char is received in UART1
		while (!UARTBufferSize(1));
		vTaskDelay(10);
		len_s = UARTBufferSize(1);
		// Read the string received and put it
		// on the variable string_serial
		UARTRead(1,string_serial,len_s);
		// add a terminator character
		string_serial[len_s] = '\0';
		strlen(string_serial);
	}
}
```

The result on a web browser if user writes the string *"changed value..."* on UART1 will be the shown above, **only when the whole web page has been reloaded:**



NOTE: *if user don't refreshes the web page, the old value will be showed even if actually a new value of dynamic variable "string_serial" is assigned.*

The assignment of the right value at dynamic `~string_serial~` as the real value of "`char string_serial[20]`" is done by a file HTTPApp.c.

The content of *"HTTPApp.c"* file is written above:

```
/****************************************************************************
  SECTION    Include
****************************************************************************/

#include "TCPIP Stack/TCPIP.h"
#if defined(STACK_USE_HTTP2_SERVER)


extern char string_serial[];

/****************************************************************************
  SECTION    Define
****************************************************************************/
#define __HTTPAPP_C


/****************************************************************************
  SECTION    Authorization Handlers
****************************************************************************/


/*****************************************************************************
  FUNCTION   BYTE HTTPNeedsAuth(BYTE* cFile)

  This function is used by the stack to decide if a page is access protected.
  If the function returns 0x00, the page is protected, if returns 0x80, no
  authentication is required
*****************************************************************************/
#if defined(HTTP_USE_AUTHENTICATION)
BYTE HTTPNeedsAuth(BYTE* cFile)
{
//If you want to restrict the access to some page, include it in the folder "protect"
	//	here you can change the folder, or add others
	if(memcmppgm2ram(cFile, (ROM void*)"protect", 7) == 0)
		return 0x00;		// Authentication will be needed later

	// You can match additional strings here to password protect other files.
	// You could switch this and exclude files from authentication.
	// You could also always return 0x00 to require auth for all files.
	// You can return different values (0x00 to 0x79) to track "realms" for below.

	return 0x80;			// No authentication required
}
#endif

/*****************************************************************************
  FUNCTION   BYTE HTTPCheckAuth(BYTE* cUser, BYTE* cPass)

  This function checks if username and password inserted are acceptable

  ***************************************************************************/
#if defined(HTTP_USE_AUTHENTICATION)
BYTE HTTPCheckAuth(BYTE* cUser, BYTE* cPass)
{
```

```
	if(strcmppgm2ram((char *)cUser,(ROM char *)"admin") == 0
		&& strcmppgm2ram((char *)cPass, (ROM char *)"flyport") == 0)
		return 0x80;		// We accept this combination

	// You can add additional user/pass combos here.
	// If you return specific "realm" values above, you can base this
	//   decision on what specific file or folder is being accessed.
	// You could return different values (0x80 to 0xff) to indicate
	//   various users or groups, and base future processing decisions
	//   in HTTPExecuteGet/Post or HTTPPrint callbacks on this value.

	return 0x00;			// Provided user/pass is invalid
}
#endif

/****************************************************************************
  SECTION	GET/POST Form Handlers
****************************************************************************/

/****************************************************************************
  FUNCTION	HTTP_IO_RESULT HTTPExecuteGet(void)

  This function processes every GET request from the pages.
*****************************************************************************/
HTTP_IO_RESULT HTTPExecuteGet(void)
{


	return HTTP_IO_DONE;
}


#ifdef HTTP_USE_POST
/****************************************************************************
  FUNCTION	HTTP_IO_RESULT HTTPExecutePost(void)

  This function processes every POST request from the pages.
*****************************************************************************/
HTTP_IO_RESULT HTTPExecutePost(void)
{


	return HTTP_IO_DONE;
}
#endif

void HTTPPrint_string_serial(void)
{
	TCPPutString(sktHTTP,(BYTE*) string_serial);
}

#endif
```

As it can be seen, there are two main operations in this file, and them are the declaration of

extern char string_serial[];

and the function

void HTTPPrint_string_serial(void)

that execute a ***TCP write*** to communicate the value of *string_serial* at the client

? **QUESTION: Is there a way to update values without the need of refresh the whole page?**
The answer lies on the using of AJAX.

AJAX in action

AJAX is used to automatically update values in webpages without the need of continue refreshes of the overall HTML page.
Using AJAX the request of data to Flyport will be prompted directly by webpage, and browser don't have to refresh all the page any more, since data became really dynamic and with automatic refreshes of only the data needed to be updated.

Here is an *example of a serial to webpage string*, but **with the using of AJAX scripts**. In this case, the *index.htm* file is not the only one necessary for webserver source, but to use AJAX the folder structure is the following:

- *index.htm*
- *status.xml*
- *leds.cgi*
- *mchp.js*
- *header.inc*
- *footer.inc*
- *style.css*
- *images/openpicus_logo_blog.png*
- *images/arrow.gif*
- *images/bar.gif*
- *images/baractive.gif*
- *images/barul.gif*
- *images/bg.gif*
- *images/h3bg.gif*
- *images/intro.jpg*
- *images/leftintrobg.gif*

Take a look of the browser render of this new web page:

The images are not necessary, but are used to give a better looking; “header.inc”, “footer.inc” and “style.css” are used by “index.htm”.

In ***status.xml*** it can be seen the dynamic variable `<testing>`**`~test~`**`</testing>` associated to “testing” id of the ***index.htm*** code

```
<p>String:<br />
<span id="testing">?</span></p>
```

The value is loaded and updated every 10ms, so when webpage loads the “testing” span, it automatically loads the dynamic variable value **`~test~`** that is updated by the function

```
// Update string value
document.getElementById('testing').innerHTML =
                                        getXMLValue(xmlData, 'testing');
```

This specific function is part of the file *mchp.js* scripts, and are the responsible of all data updates since it gets the value by accessing of xml files.

The updated value is given by a specific *callback* function of Flyport's HTTP firmware, and the result is wrote on the *xml* file.

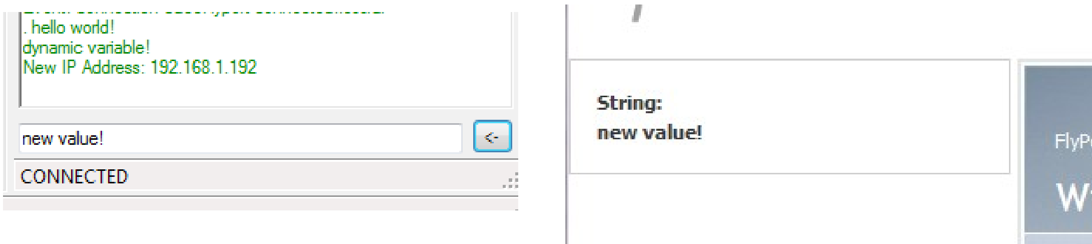
Any time Flyport handles a dynamic variable, it executes the related callback function, for example ***`HTTPPrint_test()`***. All the callback functions start with “*HTTPPrint_” plus* then the name of the

variable (test in this case). When the Flyport sends the file “status.xml” to the browser, it finds the dynamic variables inside and execute the callback functions.
Asking for a dynamic variable is just a way to make Flyport to execute a callback function.

Flyport's HTTP task updates the values with the function:

```
void HTTPPrint_test()
{
	TCPPutString(sktHTTP,(BYTE *)string_serial);
}
```

To see the result of the using of AJAX in our page, it is sufficient to write a string like “new value!” on UART1 using a simple serial terminal (like the one provided by OpenPicus IDE) and look to the web page changes!



That is the way AJAX permits Flyport to update automatically the variables displayed in web browser. It can be used to monitor Flyport's hardware status, external devices like sensors, or anything else the firmware can handle.

SNTP Client

The ***SNTP*** service is a *Simple Network Time Protocol*, a simplified version for embedded devices of NTP protocol. With the using of this feature, Flyport can download from the internet network very precise time informations from dedicated time servers.
With the using of this feature, Flyport can be always in sync with the “real world” time. In fact, every embedded system has it's own timers and counters, but them are always related to a unspecific starting time. Also the internal *Real Time Clock Calendar (**RTCC**)* needs to be tuned to a starting point, and this settings should be done by user task.
With SNTP Flyport can sync RTCC with real time, or send emails at defined “timestamps” or display to a webpage or a display a constantly updated clock value...

SNTP functionalities

The result of a SNTP request will be a number of 32 bit (a DWORD) that is the count of seconds from the 1st January 1970 to the current time. So, 0 is the value related at *1st January 1970 00h:00min:00sec*, 3 is the value related at *1st January 1970 00h:00min:03sec*, and so on.
Use this kind of value could be very uncomfortable for user application, so it is almost always needed to use some converting functions.

To convert the DWORD value to a user friendly structure OpenPicus framework needs 3 passages:

DWORD → time_t → struct tm

Using this 3 step conversion, all the timing values are easy accessible and well formatted as *minutes, seconds, days, months, etc...*

Also an external file called “heap.s” is needed to use the conversion functionalities. The content of *heap.s* is the following:

```
.section my_heap,heap
.space 0x07D0
```

a very few commands, but necessary for a correct compilation of the SNTP helpers functions.

This file should be included in project using the “*External Libs”* tool.

NOTE: *If the “external lib” tool is used, the “heap.s” file should be placed outside the project folder*

SNTP usage example

A very short usage example of SNTP features is shown here:

```
#include "taskFlyport.h"
#include "time.h"

// Define to have debug console for SNTP messages
#define _UART_DEBUG_SNTP_

char string_serial[50]="waiting...";
time_t now;
struct tm *ts;
DWORD epoch=0;
DWORD epochtime=0xA2C2A;
char dateUTC[100];
char dateUTC1[50];

// to properly set GMT by adding or removing
// the hours for GMT zone (for example Rome = +1 or +2 GMT)
// negative values are supported too...
int GMT_hour_adding = 2;

void FlyportTask()
{
 WFConnect(WF_DEFAULT);
 while (WFStatus != CONNECTED);
 UARTWrite(1,"Flyport connected... hello world!\r\n");

 vTaskDelay(200);

 UARTWrite(1,string_serial);
 while(epoch<epochtime)
 {
       vTaskDelay(50);
       epoch=SNTPGetUTCSeconds();
 }
 UARTWrite(1, "done!\r\n");


 while(1)
 {
       vTaskDelay(20);

       epoch=SNTPGetUTCSeconds();
       now=(time_t)epoch;
       ts = localtime(&now);

       // GMT adding 2 hours test
       ts->tm_hour = (ts->tm_hour + GMT_hour_adding);
       // Correct if overflowed hour 0-24 format
       if(ts->tm_hour > 24)
       {
                   ts->tm_hour = ts->tm_hour - 24;
```

```
        }
        else if(ts->tm_hour < 0)
        {
              ts->tm_hour = ts->tm_hour +24;
        }
        strftime(dateUTC1, sizeof(dateUTC1), "%Y-%m-%d / %H:%M.%S", ts);
        sprintf(string_serial, "%s", dateUTC1);

        #ifdef _UART_DEBUG_SNTP_
        if(UARTBufferSize(1) > 0)
        {
              sprintf(dateUTC,"%s\r\n",dateUTC1);
              UARTWrite(1,dateUTC);
              UARTFlush(1);
        }
        #endif

  }
}
```

In this example it can be seen the most important passages of SNTP usage.
First of all, it must be included the *"time.h"* compilers library, that permits the handle of **time_t** variables and **struct tm**.

After the time.h including, there are the declarations of the used variables:

```
time_t now;
struct tm *ts;
DWORD epoch=0;
DWORD epochtime=0xA2C2A;
```

epoch and **epochtime** are two 32bit integer.
The first one stores the value returned by the

```
epoch=SNTPGetUTCSeconds();
```

the second have a value higher than a certain period since the *SNTPGetUTCSeconds()* function does return the correct value after some time passed. To fix this problem there is the

```
while(epoch<epochtime) loop
```

checks when the SNTP is sending the right value, and not a too small one.

Once the setting time is elapsed, and application passed the first while loop, it enters on the infinite while(1) and update the variables every time the loop is repeated.

```
epoch=SNTPGetUTCSeconds();
now=(time_t)epoch;
ts = localtime(&now);
```

In this way, it can be used the **ts** variable and parse all the values using the

```
strftime(dateUTC1, sizeof(dateUTC1), "%Y-%m-%d / %H:%M.%S", ts);
```

that formats the values in the format desired, copying the result string to "*dateUTC1*"

The string format can be customizable at user preferences, for example it can be changed as:

```
"%H:%M.%S of the day %d of month %m of the year %Y"
```

QUESTION: How can be changed the "time-zones" references for GMT hour setup?
In the previous example there is the variable "`int GMT_hour_adding`" that permits the setting of different time-zones. Changing this variable (also at runtime by user is possible) it can be easily changed time-zones adding or subtracting hours from the "GMT+0" zone (Greenwich Mean Time synchronized hour)

Advanced Features

In this chapter are shown some advanced features of Open Picus Framework. It is suggested to first learn the standard functionalities of Framework and to approach in a second time those features as them can be locking for the overall system.

RTCC peripheral module

The **Real Time Clock Calendar** is a PIC24F module that permits to take track of the time with the using of the secondary oscillator. This feature permits to handle timed alarm interrupts without any polling by user task. RTCC can be seen as an advanced alarm clock, because it's alarm can executed not only every day (like any other standard alarm clocks) but with different check configurations.

RTCC structure

OpenPicus framework has integrated a RTCC helper library to permit users to don't mess with PIC internal registers, and reduce the functions calling to setup RTCC module.
To use this library it is needed to *import* the *"LIB – RTCC.zip"* content files with the IDE tool, and include the header with the statement *#include "rtcc.h"*.

When right imported the library, and included the header for functions and variable declarations, Framework can use the ***t_RTCC*** function type, that can store all the calendar-time informations.

t_RTCC rtcc;

with the declaration of the *rtcc* variable, there will be created a structure with those BYTE elements:

- rtcc.year (only last two digits)
- rtcc.month (1 to 12)
- rtcc.day (1 to 31)
- rtcc.dweek (day of the week, Sunday = 0, Saturday = 6)
- rtcc.hour (0 to 23)
- rtcc.minutes (0 to 59)
- rtcc.sec (0 to 59)

For **example** to assign the values related at the date **"June 15, 2011 (Wednesday), hour 16:20.30"** to a defined structure it can be used this sequence of commands:

```
t_RTCC *mytime;
mytime → year     = 11;
mytime → month    = 6;
mytime → day      = 15
mytime → dweek    = 4;
mytime → hour     = 16;
mytime → min      = 20;
mytime → sec      = 30;
```

This sequence will prepare the t_RTCC structure *"mytime"* to be used to store values to hardware registers, using the helper functions of library.

RTCC functions

Once the setup of user's t_RTCC structure is finished, the *information exchange* between this one and RTCC module register can be done using the following functions:

To Write parameters to Registers → RTCCWrite(t_RTCC* rtcc);
This function writes all the data stored in rtcc variable inside the RTCC module's internal registers.
This function will also start RTCC module counting functionalities (the clock is started).
Parameters:
rtcc: the t_RTCC user's variable

To Read parameters from Registers → RTCCRead(t_RTCC* rtcc);
This function reads all the data stored in RTCC module's internal registers and copy them to user's variable.
Parameters:
rtcc: the t_RTCC user's variable

To Setup Alarm → RTCCSetAlarm(t_RTCC* rtcc,
int repeats,
BYTE mask);
This function configures RTCC alarm function. Since RTCC can be considered as an advanced alarm clock, its alarm features can be setted with more configurations than a normal alarm clock.
The repetitions can be from only one alarm to infinite.
Also a different mask can be used, so alarm could be executed every day like every alarm clocks, but also every week, every month, every minutes, etc...
Parameters:
rtcc: the t_RTCC user's variable
repeats: the number of repetitions
can be used:
REPEAT_NO
REPEAT_INFINITE
or a number form 0 to 256(infinite)
mask: the check masking options
can be used:
EVERY_HALF_SEC
EVERY_SEC
EVERY_TEN_SEC
EVERY_MIN
EVERY_TEN_MIN
EVERY_HOUR
EVERY_DAY
EVERY_WEEK
EVERY_MONTH
EVERY_YEAR

The meaning the **mask** parameter is following explained with some examples:

EVERY_YEAR means that alarm will be executed every time the number before the "year" BYTE will be the same. In this way if it is setted a RTCC alarm of "*June 13, 1976 hour 16:30.15*", every J**une 13, hour 16:30.15** the alarm will be executed. If the Flyport is leaved turned on, the alarm will be executed once a year.

EVERY_HOUR means that alarm condition will be encountered once a hour, so the check of alarm condition is done analysing the values "min" and "sec". If it is setted a RTCC alarm of "*June 13, 1976 hour 16:30.15"*, **every time the clock is in the format 30 minutes and 15 seconds** the alarm condition is satisfied.

To Activate/Deactivate alarm → RTCCRunAlarm(BYTE run);
This function activates or deactivates the alarm
Parameters:
run: 0 to turn off, 1 to turn on

NOTE: *to have a complete example on RTCC usage, please download the RTCC related Application Note at www.openpicus.com that includes library and example project.*

The energy saving modes

Open Picus Framework permits the using of advanced energy saving methods. This is a advanced feature, as the Flyport's normal operations and functionalities will be different, and all the TCP/IP related functions feedback will be not reliable since them values could be not real. As a general rule, all the functionalities described in the chapter *"Using the TCP/IP stack"* should not be used since are all TCP/IP dependant.

There are two ways to reduce power consumption:

- Turn OFF Wi-Fi transceiver only (**Hibernate** mode)
- Turn OFF Wi-Fi transceiver and put in SLEEP state the microcontroller (**Sleep** mode)

Both modes turns OFF the Wi-Fi transceiver to save power, but the Sleep mode saves more power stopping the Microcontroller's jobs. Only RTCC and external interrupts can wake up Microcontroller once in sleep.

Special considerations:
All the TCPSockets and UDPSockets must be reinitialized after a wake up from hibernate or sleep mode. This must be done every time the Flyport wakes up and for every Sockets used, or the Flyport will fail its tasks executions and will be definitely locked!
Please, pay very special attention at socket handle as it is the most delicate software passage for power saving modes.

Hibernate mode

Activating the Hibernate mode, the Wi-Fi transceiver will be turned OFF, the Open Picus framework will kill the TCP task, and only the user task will be executed. This is not the maximum power saving feature, but PIC is still powered.

The functions described in *"Controlling the Flyport hardware"* chapter, and "*RTCC peripheral module"* can be used without any kind of warning as them are only PIC dependant.

To Activate Hibernate Mode → WFHibernate();
This function activates the Hibernate mode, so turn OFF the Wi-Fi transceiver

To Deactivate the Hibernate Mode → WFOn();
This functions turns ON the Wi-Fi transceiver again

NOTE: *this function does NOT reconnect Flyport to Wi-Fi network automatically. User should handle the connection on Wireless LAN once the transceiver is enabled again.*

Sleep mode

Activating the Sleep mode the Wi-Fi transceiver is turned OFF, and also PIC enters in a low power mode to save energy. This mode however lock almost all the power calculations of microcontroller, so only external events can wake up Flyport from this state.

The external events that can wake up Flyport are 2:

- **RTCC alarm**
- **External Interrupts (up to 3 differents)**

Both the described events launches special Interrupt Service Routines (ISR) that permits the microcontroller to wake up from a sleep condition, it is not important of what the ISR related function do, the important is that those functions are exited from external events not related to PIC functionalities (the RTCC alarm could be seen as external event since it uses secondary oscillator and is independent from microcontroller).

To Activate Sleep Mode → **WFSleep();**
This function activates the Sleep mode, so turn OFF the Wi-Fi transceiver and stops microcontroller working.

To Deactivate Sleep Mode
Only *RTCC* and *external interrupts* can deactivate the sleep mode.

QUESTION: How can I use those features in my application? How can I set a time to wakeup my Flyport?
In the next example there is the usage of both RTCC and External Interrupts to wakeup Flyport from sleep state at defined timestamps or at user needs with the change of pin state.

Energy saving usage example

Here is a simple power saving example, using Serial Commands to pass from normal operation to hibernate or sleep, and RTCC or External interrupt 2 to wakeup from sleep mode.

```
#include "taskFlyport.h"
#include "rtcc.h"

char input[20];
char rtccString[50];

extern BOOL alarmflag;

t_RTCC myrtcc;
t_RTCC myalarm;
t_RTCC nowrtcc;


void external_interrupt_function()
{
	// Reset interrupt
	IOPut(o4,on);
	UARTWrite(1,"Wakeup!\r\n");
}


void FlyportTask()
{
	vTaskDelay(200);
	UARTWrite(1,"\r\nPower saving mode test...\r\n");

	// Set external interrupt
	IOInit(p5,inup);
	IOInit(p5, EXT_INT2);
	INTInit(2, external_interrupt_function, 1);
	INTEnable(2);

	// Set RTCC
	myrtcc.year = 11; //last two year number
	myrtcc.month = 11;
	myrtcc.dweek = 3; //sunday is 0
	myrtcc.day = 2;
	myrtcc.hour = 16;
	myrtcc.min = 23;
	myrtcc.sec = 1;
	// Write settings on internal registers
	RTCCWrite(&myrtcc);
	// Create alarm configuration
	myalarm = myrtcc;
	myalarm.sec = myalarm.sec + 20;
	// Set Alarm configuration to internal registers
	RTCCSetAlarm(&myalarm, REPEAT_INFINITE, EVERY_TEN_SEC);
	// Active alarm
```

```
RTCCRunAlarm(1); // 1 turn on, 0 turn off

while(1)
{
      // Check RTCC alarm flag
      if(alarmflag == 1)
      {
            alarmflag = 0;
            vTaskDelay(1);
            UARTWrite(1, "ALARM!!!\r\n");
      }
      // Check UART commands
      if (UARTBufferSize(1) > 1)
      {
            vTaskDelay(50);
            char uread[257];
            int toread = UARTBufferSize(1);
            UARTRead(1,uread, toread);
            uread[toread+1]='\0';

            // Command parsing
            if (strstr(uread,"off")!=NULL)
            {
                  UARTWrite(1,"WiFi OFF\r\n");
                  WFHibernate();
            }
            else if (strstr(uread,"sleep")!=NULL)
            {
                  UARTWrite(1,"Flyport is sleeping... \r\n");
                  WFSleep();
            }
            else if (strstr(uread,"on")!=NULL)
            {
                  UARTWrite(1,"Wi-Fi transceiver activation...\r\n");
                  WFOn();
            }
            else if (strstr(uread,"up")!=NULL)
            {
                  UARTWrite(1,"Connecting...\r\n");
                  WFConnect(WF_DEFAULT);
                  UARTWrite(1,"Connection launched...\r\n");
            }
            else if (strstr(uread,"dn")!=NULL)
            {
                  UARTWrite(1,"Disconnecting...\r\n");
                  WFDisconnect();
            }
            else if (strstr(uread,"rtcc")!=NULL)
            {
                  // Print RTCC (only if NOT IN SLEEP MODE)
                  RTCCRead(&nowrtcc);     // get the current time
                  UARTWrite(1, "RTCC state:\r\n");
                  sprintf(rtccString, "%d/%d/%d\r\n", nowrtcc.hour,
                                          nowrtcc.min, nowrtcc.sec);
                  UARTWrite(1, rtccString);
            }
```

```
                UARTFlush(1);
            }
        }
}
```

In this example it can be seen all the passages that should be done to use the power saving functionalities. Using UART commands user can control every single step to switch between *normal operation, sleep mode, Wi-Fi off, connection, disconnection and print RTCC actual state.*
To pass from one state to another some according should be done, for example if user tries to connect to Wi-Fi network when transceiver is turned OFF Flyport will attempt to open a connection, but no warnings messages will be shown on UART.

QUESTION: Are there some defined sequences to respect to use Power Saving Modes?
There are the command sequences that could be used without any problems:

To switch to **transceiver off** user should use:
"off"

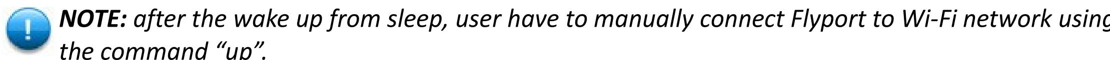
To switch to **normal mode** user should use:
"on"

NOTE: *with this command, even if Flyport was connected before, user should connect again to Wi-Fi network using "up"*

The switch between normal state and transceiver off can be done with the commands "on" and "off", with attention to connect again Flyport to Wi-Fi profileafter turning on the transceiver

To switch to **sleep mode** user should use:
"sleep"

NOTE: *with this command, only interrupts can wake up Flyport.*
In this specific example there are RTCC alarm and External interrupt 2 connected to pin 5 with pull up resistor; to wake up Flyport to sleep state user can wait for RTC ALARM or connect pin 5 to Ground (and excite the external interrupt).



NOTE: *after the wake up from sleep, user have to manually connect Flyport to Wi-Fi network using the command "up".*

To check the working (wake up or sleep) of **microcontroller** user can use:
"rtcc"

NOTE: *This command makes Flyport to print the RTCC actual state on UART, but works only when microcontroller is active; Flyport will not reply when in sleep state!*

The switch between normal state and sleep can be done with the command "sleep" and waiting for RTCC or changing state of interrupt pin, with attention to connect again Flyport to Wi-Fi profile after the wakeup.

⚠ ***Warning**: Pay attention also at the sockets reinitializzation used in the application. All the TCP sockets used in user task should be reinitialized to "INVALID_SOCKET" value before open them again; same for UDP Sockets, that should be reinitialized to "0". This is a necessary passage, and it is suggested to do it every time the user turn on again the transceiver, and before any connection to network.*

Rome, Italy • +39.06.92916378 • www.openpicus.com