SquareWear Programming Reference 1.0

Oct 10, 2012

Content:

---

1. Overview

SquareWear v1.1 is based on Microchip's PIC18F14K50 microcontroller running at 12MHz. It has 16KB program space, 768B RAM, and 256B EEPROM. The software program is built on MPLAB X IDE with C18 compiler for PIC18. The **SquareWear library** builds upon C18's peripheral library (**plib**), which comes with the installation of C18 compiler. The SquareWear library implements a number of high-level functions to accommodate common prototyping need and abstract away low-level details. If needed, you can call **plib** functions directly to access all available peripheral functions (please refer to C18's documentation for details).

Additional Resources:

- Microchip MPLAB C18 Peripheral Library Documentation (installation folder)
- Microchip MPLAB C18 User Guide
- Microchip MPLAB IDE User Guide (installation folder)
- Microchip MPLAB C18 Getting Started Guide (installation folder)

---

2. Basic Data Types

The basic data types are:

- **`char`**: signed, 8-bit
- **`byte`**: unsigned, 8-bit (equivalent to `unsigned char`)
- **`int`**: signed, 16-bit
- **`uint`**: unsigned, 16-bit (equivalent to **`unsigned int`**)
- **`long`**: signed, 32-bit
- **`ulong`**: unsigned, 32-bit (equivalent to **`unsigned long`**)
- **`float`**: 32-bit floating point

As in standard C programming language, you can define structure, union, bit field, array etc. Please refer to the MPLAB C18 User Guide for the complete list of data types.

---

3. Pin Functions

SquareWear 1.1 has 8 general I/O pins, and 4 additional output-only pins that function as power sinks. Among the 8 general I/O pins, 6 can be used as analog input; in addition, 2 among them can be used for external interrupts, 1 for hardware PWM, 2 for I2C (SDA/SCL), and 2 for USART (TX/RX). Software PWM is supported on all Port C pins. The on-board LED is internally wired to pinC7, and the on-board push-button is internally wired to pinA3 (input-only pin). The push-button can be used as a general-purpose button; it is also used to trigger software reset and enter programming mode following reset.

Pin Name and Function Table:

Pin Name	*Basic Function*	*Analog Input*	*Other Functions*	*Soft. PWM*	*Notes*
pinC0	general I/O	**pinADC4**	-	yes	
pinC1	general I/O	**pinADC5**	ext. interrupt INT1	yes	
pinC2	general I/O	**pinADC6**	ext. interrupt INT2	yes	
pinC3	general I/O	**pinADC7**	-	yes	
pinC4	*power sink only*	-	-	yes	
pinC5	*power sink only*	-	hardware PWM	yes	
pinC6	*power sink only*	-	-	yes	
pinC7	*power sink only*	-	-	yes	on-board LED
pinB4	general I/O	**pinADC10**	SDA (I2C)	-	
pinB5	general I/O	**pinADC11**	RX (USART)	-	
pinB6	general I/O	-	SCL (I2C)	-	
pinB7	general I/O	-	TX (USART)	-	
pinA3	*input only*	-	-	-	on-board push-button

Note that since some pins can be used for analog input, you need to refer to the analog name when accessing the analog functions. For example, pinC0 is also pinADC4. You should use pinC0 for accessing digital functions, and pinADC4 for accessing analog functions.

---

4. main() and initSquareWear() Functions

As in standard C, your program must have a **`main()`** function, which is the entry function to your program. Also, you must include **`SquareWear.h`**, and, at the beginning of the main function, you must call **`initSquareWear()`** first in order to properly set up SquareWear and the necessary timers. Your program generally should look like this:

```
#include “SquareWear.h”

// user functions
// .. .. ..

void main(void) {
      InitSquareWear();
      // .. .. ..
}
```

5. Digital Input / Output

```
void setModeOutput(byte pinname);
```

Set up a pin as digital output. Example: `setModeOutput(pinB4);`
Upon initialization, all port C pins will be set as output pins by default.

```
void setModeInput(byte pinname);
```

Set up a pin as digital input. Example: `setModeInput(pinB4);`

```
void setValue(byte pinname, byte value);
```

Assign a value (0 or 1) to a digital output pin. Example: `setValue(pinB4, 1);`
Note that the same function can be used to set value to a PWM output pin, as described in the next section.

```
uint getValue(byte pinname);
```

Return the value from a digital input pin. Example: `getValue(pinB4);`
Return value will be 0 or 1. Note that the same function can be used to get value from an analog input pin, as described in the section below.

Direct Access of Pin Values

In addition to using the setValue and getValue functions, you can also directly access pin values by using pin aliases. There are two types of pin aliases: one is pre-fixed with '**port**', such as **portC3**. For example,

`portC3 = 1;` is equivalent to **`setValue(pinC3, 1);`**

Similarly,

`a = portB4;` is equivalent to **`a = getValue(pinB4);`**

The other type of alias is pre-fixed with '**lat**', such as **latC3**. This should be used on **read-modify-write** operations, for example:

`latC7 = !latC7;` toggles pinC7 and is equivalent to **`setValue(pinC7, !getValue(pinC7));`**

These direct access functions are **much faster** than calling the setValue and getValue functions. **You should use them as often as you can.**

pinC7

pinC7 is internally wired to the on-board LED. Whenever it's set to high, the LED will light up.

pinA3

pinA3 is internally wired to the on-board push-button. You can check the button status by reading **portA3**. The return value is 1 if the button is released (pulled high); and the value is 0 if it's pressed. You can also use the function:

```
bool buttonPressed();
```

to check the button status. It returns 1 (logic high) if the button is pressed, and 0 (logic low) if the button is released.

pinC4 – pinC7

These four pins are internally connected to on-board MOSFETs and they function as power sinks. The typical way of

using them is to connect the positive end of a component (such as an LED) to Vdd (Source), and the negative end to a power sink pin. This way, by sending setting the pin to 1 or 0 you can turn on or turn off the component. Because of the way they are set up, these 4 pins are only used as digital output or PWM output – they do not work as input pins.

---

6. Analog Input / PWM Output

```
void setModeADC(byte pinname);
```

Set up a pin as analog input. Example: `setModeADC(pinADC4);`
As described above, here you must use the analog pin names, even though they are physically the same as the digital pins. For example, pinADC4 is physically the same as pinC7. You need to use the name pinC7 for setting it up as digital input or output, and pinADC4 for setting it up as analog input.

```
uint getValue(byte pinname);
```

Return the value from an analog input pin. Internally this function performs an analog-digital conversion (ADC) and returns a value in the range from **0 to 1023 (10-bit)**.

As described above, if the pinname is given as a digital pin name, the return value will be binary (0 or 1).

Pulse-Width Modulation (PWM) Output

PWM is a method to simulate analog output (i.e. values in between digital high and low) by producing a high-frequency digital signal and adjusting its duty cycle (i.e. the time share of digital highs). SquareWear supports software PWM output on all Port C pins (pinC0 to pinC7).

```
void setModePWM(byte pinname);
```

Set up a pin as PWM output. Example: `setModePWM(pinC7);`
Again, this only works on the eight Port C pins.

```
void setValue(byte pinname, byte value);
```

Assign a value, between **0 and 31 (5-bit)** to a PWM output pin. So the **PWM resolution is 32 grades** (0 is equivalent to digital low or 0V, and 31 is equivalent to digital high or Vdd).

The software PWM works by generating a high-frequency timer interrupt, and at each cycle adjusting the digital output based on the desired duty cycle (value). For example, for duty cycle of 50%, the digital signal will be Vdd (high) half of the time, and 0 (low) the other half of the time. So on average the output value is about 50% of Vdd.

SquareWear generate software PWM at **183Hz** frequency. This is not high, but sufficient to avoid LED flickering. If you want higher frequency PWM, you should use the hardware PWM. Hardware PWM is much faster, but a microcontroller typically only has a small number of pins that support hardware PWM. In comparison, software PWM is more flexible and can be simulated on any digital pins.

---

7. Timing, Delay, Reset, and Sleep Functions

```
ulong millis();
```

Return milliseconds that has passed since the start of the program.

```
void delayMicroseconds(int us);
void delayMilliseconds(int ms);
void delaySeconds(int seconds);
```

Delay the specified amount of microseconds, milliseconds, or seconds.

```
void openSoftwareReset();
void closeSoftwareReset();
```

Software reset is a feature where if you have pressed the on-board push-button for more than 5 seconds, it will trigger a reset and bring the microcontroller to programming mode. This feature is turned on by default. It is implemented by using a timer interrupt to periodically check the button status, so it does not interfere with or stall your main program.

```
void idleSleep();
void deepSleep();
```

These two are sleep functions that can help reduce the microcontroller's energy consumption and save power. In **idleSleep**, the mcu's clock source continues running, so the time keeping functions work as usual, and all timer interrupts are triggered as usual. The mcu will return back to normal execution upon interrupt events, such as button interrupt, timer interrupt, or external interrupts.

In **deepSleep**, the mcu's clock source will stop running, so all time-related functions will stop working. The only way to wake up the mcu is by using button interrupt or external interrupts (which do not rely on timers). This is useful to put the mcu to minimal power state. You can combine it with button interrupt or external interrupts to allow the mcu to run over a long time without draining the battery.

---

8. USB Serial Functions

SquareWear's built-in USB module can be used to create a USB Serial (CDC) port in order to perform serial communications with a computer. This does not require any external USB Serial cable.

```
void openUSBSerial();
```

Open the USB serial port. If a USB cable is plugged in, your operating system should automatically detect the USB Serial device (CDC) after this function is called.

- On Linux or Mac, you do not need any driver. The system automatically detects the device and it should appear as **/dev/ttyACMx** in Linux or **/dev/ttyusbserial.x** in Mac (where x is system assigned number).
    - In case it appears as /dev/ttyUSBx in Linux, you can unplug the USB, run '**sudo rmmod ftdi_sio**' and re-plug in the USB until it shows up as /dev/ttyACMx.
- On Windows, you need to install the CDC device driver, available in the **inf/** folder under the **uploaders/** directory. Once installed, the device should appear as COMx port (where x is a system assigned number).

```
void putsUSBSerial(char *string);
```

Output a string (array of chars) to USB Serial. Example:

```
char msg[]=”hello”;
putsUSBSerial(msg);
msg[0]='H';
putsUSBSerial(msg);
```

```
void putrsUSBSerial(const rom char *string);
```

Output a constant string to USB Serial. Example:

```
putrsUSBSerial(“hello”);
```

Note the extra 'r' in the function name. This differs from the previous function in that you must provide a constant string, not an array of chars. In MPLAB C18, any string in double quotes are compiled as constant string, and is placed in program memory space, not in RAM. In contrast, an array of chars is placed in RAM. So you need to be careful which function to use. Examples:

```
char msg[]=”Hello ”;
putsUSBSerial(msg);
putrsUSBSerial(“World!”);
```

```
void putnUSBSerial(char *string, byte n);
```

Output the first n characters of the string to USB serial.

```
void getUSBSerial(char *string, byte n);
```

Read a maximum of n characters from USB serial port. The parameter 'string' is a char array of size at least n.

```
void pollUSBSerial();
```

This function polls and processes pending USB tasks. Since it's not based on a timer interrupt, it **MUST be called as often as you can** in your program, otherwise the USB Serial connection will be lost. Example:

```
void main(void) {
      initSquareWear();
      .. .. ..
      while(1) {
            .. .. ..
            pollUSBSerial();
      }
```

Read/Write Numbers

Although the USB Serial functions cannot directly read or write numbers, you can use standard C's functions (from <**stdlib.h**>) to convert numbers to and from strings. For example, **itoa(number, string)** converts an integer to string, and **atoi(string)** does the reverse. Similarly, **ltoa** and **atol** handle long integer and string conversions; **atof** handles string to double conversion. There is no **ftoa** function, but you can use the **sprinf()** function (from <**stdio.h**>) instead.

Serial Monitor

In order to perform serial communication, you need to run a serial monitor on your computer. First of all, you can use the serial monitor built-in to **Arduino** or **Processing** – they are both cross-platform. **Alternatively**,

- On Linux, you can use **putty** or **gtkterm**. The typical serial port number is /dev/ttyACM0 but the actual port number may be different. You need to run the serial monitor in **sudo**, or alternatively add the device vendor and id (**04d8:000a**) to your **/etc/udev/rules.d/**
- On Mac, you can use **screen** or **stty**. The typical serial port number is /dev/ttyusbserialxxx.
- On Windows, you can use **putty**. The typical serial port number is COMx.

9. Timer Interrupt

SquareWear's library internally uses several timer interrupts to implement time keeping, PWM, and software reset. In addition, you can define a custom timer interrupt to perform periodic tasks without stalling your main loop.

```
void openTimerInterrupt(ulong ms, isr_handler callback);
```

This function opens custom timer interrupt function. The parameters are trigger interval (in milliseconds), and callback function. Once opened, the callback function will be executed regularly at the trigger interval. The callback function must be of type **`void function_name(void)`**. In other words, it does not take any parameter and does not return any value. Example:

```
void blink(void) {
      latC7 = !latC7;  // toggle on-board LED
}
void main(void) {
      initSquareWear();
      .. .. ..
      openTimerInterrupt(500, blink);
      while(1) {
            .. .. ..
            pollUSBSerial();
      }
```

This toggles pinC7 (hence the on-board LED) every 500 milliseconds, while the main loop performs other tasks.

Note 1:
Because an interrupt will temporarily pause the main execution path, you should keep the callback function as short as possible. A long callback function will cause significant delay on the main execution path and should be avoided.

Note 2:
If you will be changing any global variable in your callback function, you must declare the variable as **`volatile`**. This tells the compiler that the variable is subject to change inside an interrupt function. Example:

```
volatile uint counter;

void count(void) {
      counter++; // increment counter value
}
void main(void) {
      initSquareWear();
      .. .. ..
      openTimerInterrupt(500, blink);
      while(1) {
            .. .. ..
            pollUSBSerial();
      }
```

```
void closeTimerInterrupt();
```

Disable custom timer interrupt function.

10. Push-button Interrupt

Push-button interrupt allows a callback function to be executed whenever the on-board push-button has a state change, i.e. either pressed or released.

```
void openOnBoardButtonInterrupt(isr_handler callback);
```

Open on-board button interrupt. Again, callback function is a function that neither takes parameters nor return a value. Follow the same rules as timer interrupt with regard to the length of the callback function and volatile variables. As described previously, button interrupt can be used to wake up the mcu from deep sleep mode. This way, whenever the button is pressed, the mcu will resume from sleep and perform some tasks, such as displaying an LED pattern. Then after a while it goes to deep sleep again. This allows you to keep the controller on for a very long time without draining the battery.

```
void closeOnBoardButtonInterrupt();
```

Disable push-button interrupt.

---

11. External Interrupts

External interrupt is triggered by state change on a specific digital input pin. SquareWear library allows two external interrupts: **INT1** and **INT2**. They correspond to **pinC1** and **pinC2**.

```
void openExtInterrupt(byte index, byte mode, isr_handler callback);
```

Open external interrupt. The input parameters are: **index** (1 or 2), **mode** (RISING or FALLING), and the callback function. RISING means the interrupt is triggered when the input signal rises from low to high; and FALLING means the reverse. External interrupt is useful for applications such as counting the number of pulses. It can also be used to wake up the mcu from deep sleep. For example, you can use an external 1Hz square wave to wake up the mcu once per second from deep sleep.

```
void closeExtInterrupt(byte index);
```

Close external interrupt 1 or 2.

---

12. Hardware PWM

Compared to software PWM, hardware PWM has higher frequency and precision (10-bit). On the other hand, it is limited to a small number of pins. SquareWear support one hardware PWM on **pinC5**.

```
void openHardwarePWM();
Void setHardwarePWMduty(uint duty);
void closeHardwarePWM();
```

The hardware PWM runs at 11.8kHz frequency, and allows a precision of 1024 steps (10-bit). So **duty** can be anywhere from 0 to 1023.

13. EEPROM Functions

EEPROM is a common type of non-volatile memory. It is used to store and preserve values over power loss. SquareWear has 256 bytes of EEPROM.

```
void writeEEPROM(uint addr, byte data);
Void busyEEPROM();
byte readEEPROM(uint addr);
```

In these functions, **addr** is the EEPROM address (0 to 255), and the **data** to read/write is a byte (8-bit). The first time you call any EEPROM read/write function, make sure there is at least a couple hundred milliseconds of delay since the start of the program, in order for the EEPROM unit to stabilize. The **busyEEPROM** function is used to wait for an EEPROM write to complete. So you should usually call it after each **writeEEPROM**, or at least call it before the next **readEEPROM.**

---

14. USART Functions

USART is used for serial communication. In contrast to USB Serial functions, USART requires an external serial cable, such as FTDI cable, so it's less convenient. On the other hand, its implementation is much simpler and so it takes less program memory space. On SquareWear, the USART pins are **pinB5 (RX)** and **pinB7(TX)**. To use USART, you need to connect the TX and RX pins on your serial cable to the RX and TX pins respectively. Then open a serial monitor to perform the serial communication.

```
void openUSART();
Void closeUSART();
void putsUSART(char *data);
void putrsUSART(const rom char *data);
Void getUSART(char *data, byte len);
```

These functions are equivalent to the USB Serial functions. Please refer to the USB Serial functions for usage.

---

15. I2C and Flash Functions

I2C is a technique that allows two devices to communicate data to each other via only two pins (SCL and SDA). Since the I2C functions provided by the MPLAB peripheral library (plib) is sufficiently high-level, SquareWear library does not need to implement any wrapper functions for them. Please refer to the plib documentation for instructions.

The peripheral library also contains functions to read and write the microcontroller's flash memory. Unlike the Arduino, Microchip mcus allow you to write flash memory at run-time. This way you can use flash memory as non-volatile memory (in addition to EEPROM), or use it as extra (but slow) RAM space. Again, the plib functions are sufficient high-level. Please refer to the plib documentation for instructions.
