A Tutorial Guide to Programming PIC18 Microcontrollers with FlashForth.

Mechanical Engineering Report 2012/05
P. A. Jacobs
School of Mechanical and Mining Engineering
The University of Queensland.

January 7, 2013

Abstract

Modern microcontrollers provide an amazingly diverse selection of hardware peripherals, all within a single chip. One needs to provide a small amount of supporting hardware to power the chip and connect its peripheral devices to the signals of interest and, when powered up, these devices need to be configured and monitored by a suitable firmware program. These notes focus on programming a PIC18F2520 microcontroller in a simple hardware environment and provide a number of example programs (in the Forth language) to illustrate the use of some of the MCU's peripheral devices.

Contents

1 Microchip's PIC18F2520 microcontroller **3**

2 Development boards **6**

3 FlashForth **8**
3.1 Getting FlashForth and programming the MCU 9

4 Interacting with FlashForth **10**

5 Introductory examples **11**
5.1 Hello, World: Flash a LED 11
5.2 Set the cycle duration with a variable 13
5.3 Hello, World: Morse code 14

6 Read and report an analog voltage **14**

7 Counting button presses **16**

8 Scanning a 4x3 matrix keypad **17**

9 Using I^2C to get temperature measurements **19**

10 An I^2C slave example **20**

11 Speed of operation **25**

1 Microchip's PIC18F2520 microcontroller

Over the past couple of decades, microcontrollers have evolved to be cheap, powerful computing devices that even Mechanical Engineers can use in building bespoke instrumentation for their research laboratories. Typical tasks include monitoring of analog signals, sensing pulses and providing timing signals. Of course these things could be done with a modern personal computer, connected via usb to a commercial data acquisition and signal processing system but there are many situations where the small, dedicated microcontroller, requiring just a few milliamps of current, performs the task admirably and at low cost.

Modern microcontrollers provide an amazingly diverse selection of hardware peripherals, all within a single chip. One needs to provide a small amount of supporting hardware to power the chip and connect its peripheral devices to the signals of interest and, when powered up, these devices need to be configured and monitored by a suitable firmware program. These following sections provide an introduction to the details of doing this with a Microchip PIC18F2520 microcontroller, programmed with the FlashForth interpreter.

Microchip's PIC18 microcontroller units (MCUs) all have the same core, *i.e.* same instruction set and memory organisation. Your selection of which MCU to actually use in your project can be based on a couple of considerations. If you are on a tight budget and will be making many units, choose an MCU with just enough functionality, however, if convenience of development is more important, choose one with "bells and whistles". For this tutorial guide, we will value convenience and so will work with the PIC18F2520-I/SP which has:

- a nice selection of features, including a serial port, several timers and an analog-to-digital converter. See the feature list and the block diagram of the MCU on the following pages.
- a 28-pin narrow DIL package, which is convenient for prototyping and has enough I/O pins to play without needing very careful planning.
- a pinout is shown at the start of the Microchip datasheet (book) on the PIC18F2420 [1]. You will be reading the pages of this book over and over but we include the following couple of pages togive an overview.
- an internal arrangement that is around an 8-bit data bus.
- the "Harvard architecture" with separate paths and storage areas for program instructions and data.

We won't worry too much about the details of the general-purpose registers, the internal static RAM or the machine instruction set because we will let the Forth interpreter handle most of the details, however, memory layout, especially the I/O memory layout is important for us as programmers. The peripheral devices are controlled and accessed via registers in the data-memory space.

MICROCHIP

PIC18F2420/2520/4420/4520

28/40/44-Pin Enhanced Flash Microcontrollers with 10-Bit A/D and nanoWatt Technology

Power Management Features:

- Run: CPU on, Peripherals on
- Idle: CPU off, Peripherals on
- Sleep: CPU off, Peripherals off
- Ultra Low 50nA Input Leakage
- Run mode Currents Down to 11 μA Typical
- Idle mode Currents Down to 2.5 μA Typical
- Sleep mode Current Down to 100 nA Typical
- Timer1 Oscillator: 900 nA, 32 kHz, 2V
- Watchdog Timer: 1.4 μA, 2V Typical
- Two-Speed Oscillator Start-up

Flexible Oscillator Structure:

- Four Crystal modes, up to 40 MHz
- 4x Phase Lock Loop (PLL) – Available for Crystal and Internal Oscillators
- Two External RC modes, up to 4 MHz
- Two External Clock modes, up to 40 MHz
- Internal Oscillator Block:
  - Fast wake from Sleep and Idle, 1 μs typical
  - 8 use-selectable frequencies, from 31 kHz to 8 MHz
  - Provides a complete range of clock speeds from 31 kHz to 32 MHz when used with PLL
  - User-tunable to compensate for frequency drift
- Secondary Oscillator using Timer1 @ 32 kHz
- Fail-Safe Clock Monitor:
  - Allows for safe shutdown if peripheral clock stops

Peripheral Highlights:

- High-Current Sink/Source 25 mA/25 mA
- Three Programmable External Interrupts
- Four Input Change Interrupts
- Up to 2 Capture/Compare/PWM (CCP) modules, one with Auto-Shutdown (28-pin devices)
- Enhanced Capture/Compare/PWM (ECCP) module (40/44-pin devices only):
  - One, two or four PWM outputs
  - Selectable polarity
  - Programmable dead time
  - Auto-shutdown and auto-restart

Peripheral Highlights (Continued):

- Master Synchronous Serial Port (MSSP) module Supporting 3-Wire SPI (all 4 modes) and I^2C™ Master and Slave modes
- Enhanced Addressable USART module:
  - Supports RS-485, RS-232 and LIN/J2602
  - RS-232 operation using internal oscillator block (no external crystal required)
  - Auto-wake-up on Start bit
  - Auto-Baud Detect
- 10-Bit, up to 13-Channel Analog-to-Digital (A/D) Converter module:
  - Auto-acquisition capability
  - Conversion available during Sleep
- Dual Analog Comparators with Input Multiplexing
- Programmable 16-Level High/Low-Voltage Detection (HLVD) module:
  - Supports interrupt on High/Low-Voltage Detection

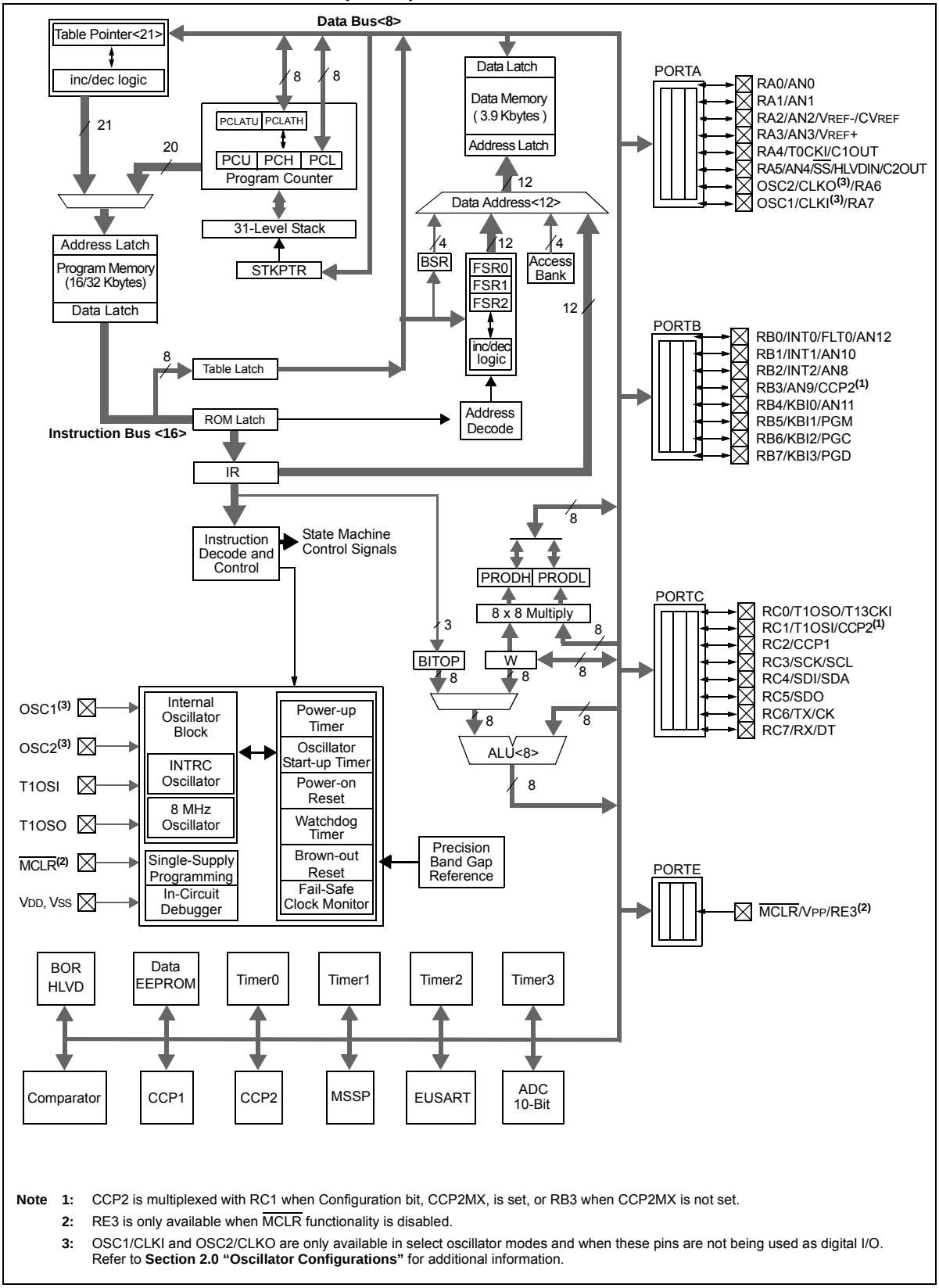
Special Microcontroller Features:

- C Compiler Optimized Architecture:
  - Optional extended instruction set designed to optimize re-entrant code
- 100,000 Erase/Write Cycle Enhanced Flash Program Memory Typical
- 1,000,000 Erase/Write Cycle Data EEPROM Memory Typical
- Flash/Data EEPROM Retention: 100 Years Typical
- Self-Programmable under Software Control
- Priority Levels for Interrupts
- 8 x 8 Single-Cycle Hardware Multiplier
- Extended Watchdog Timer (WDT):
  - Programmable period from 4 ms to 131s
- Single-Supply 5V In-Circuit Serial Programming™ (ICSP™) via Two Pins
- In-Circuit Debug (ICD) via Two Pins
- Wide Operating Voltage Range: 2.0V to 5.5V
- Programmable Brown-out Reset (BOR) with Software Enable Option

Device	Program Memory		Data Memory		I/O	10-Bit A/D (ch)	CCP/ ECCP (PWM)	MSSP		EUSART	Comp.	Timers 8/16-Bit
	Flash (bytes)	# Single-Word Instructions	SRAM (bytes)	EEPROM (bytes)				SPI	Master I^2C™			
PIC18F2420	16K	8192	768	256	25	10	2/0	Y	Y	1	2	1/3
PIC18F2520	32K	16384	1536	256	25	10	2/0	Y	Y	1	2	1/3
PIC18F4420	16K	8192	768	256	36	13	1/1	Y	Y	1	2	1/3
PIC18F4520	32K	16384	1536	256	36	13	1/1	Y	Y	1	2	1/3

PIC18F2420/2520/4420/4520

FIGURE 1-1: PIC18F2420/2520 (28-PIN) BLOCK DIAGRAM



Note 1: CCP2 is multiplexed with RC1 when Configuration bit, CCP2MX, is set, or RB3 when CCP2MX is not set.

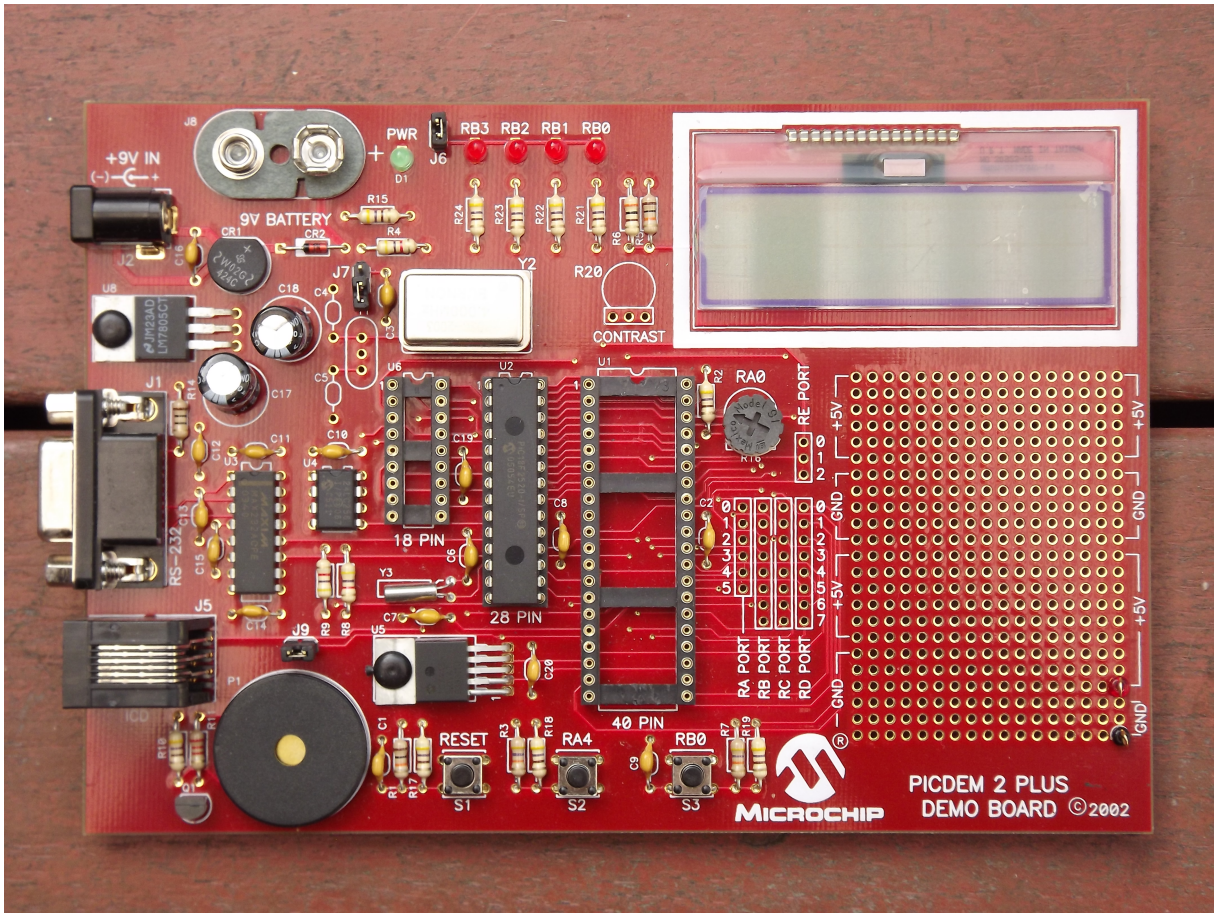
2: RE3 is only available when MCLR functionality is disabled.

3: OSC1/CLKI and OSC2/CLKO are only available in select oscillator modes and when these pins are not being used as digital I/O. Refer to **Section 2.0 "Oscillator Configurations"** for additional information.

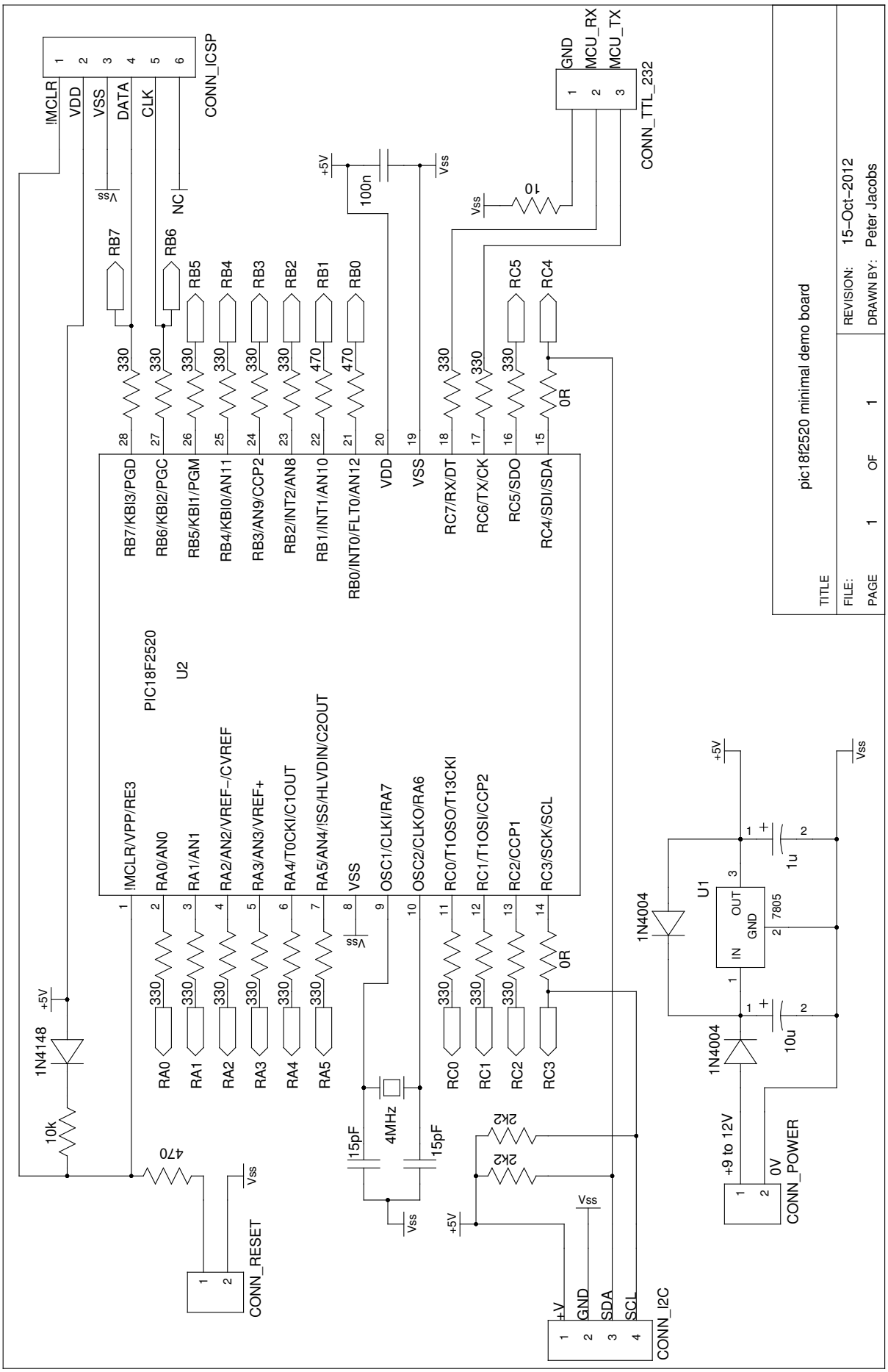
2 Development boards

This tutorial is based around simple support hardware for the MCU. If you don't want to do your own soldering, Microchip's PICDEM 2 PLUS demonstration board is a convenient way to get your hardware up and going.

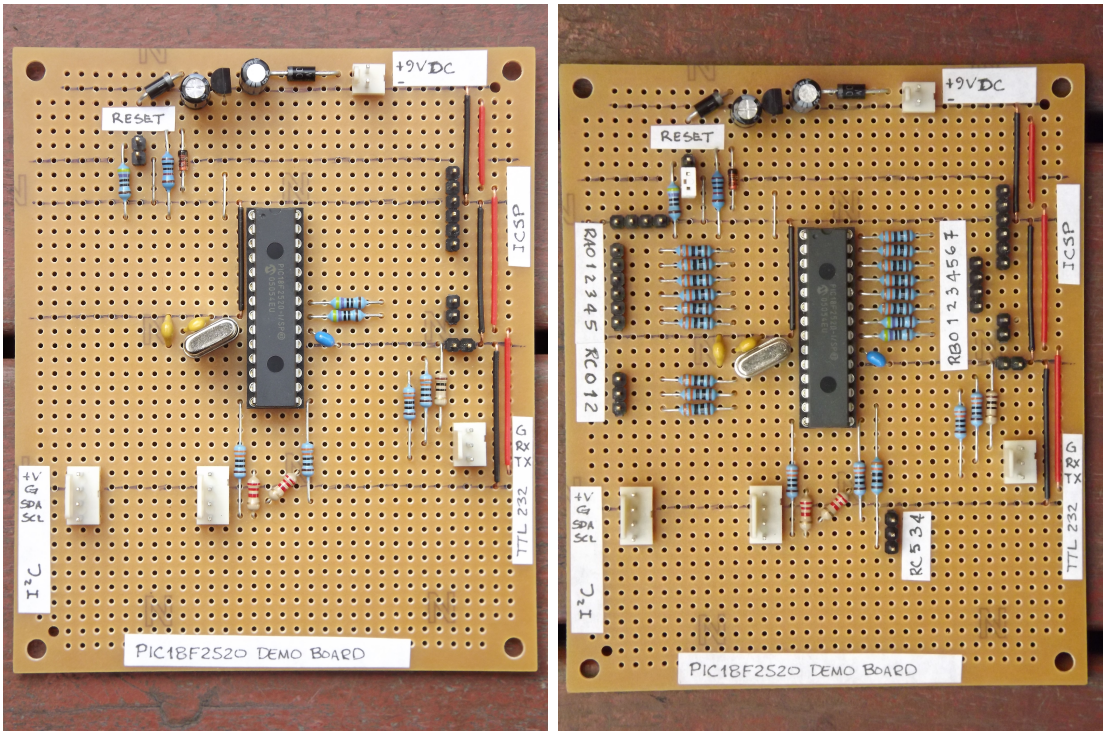
Here is a picture of PICDEM 2 PLUS with PIC18F2520 (U2) in the 28-pin socket. We'll make use of the serial RS-232 interface (MAX232ACPA, U3) to both program Forth application and to communicate with running applications. Other conveniences include on-board LEDs, switches, a potentiometer (RA0) and I^{2}C devices, such as a TC74 temperature sensor (U5), just below the MCU and a 24LC256 serial EEPROM (U4). Initial programming of the FlashForth system into the MCU can be done via jack J5 with a Microchip IDC3, PICkit3, or similar.



If you want a homebrew system, you can build a minimal system on strip-board that works well. One of the nice things about such a strip-board construction is that you can easily continue construction of your bespoke project on the board and, with careful construction, your prototype can provide years of reliable service. The schematic diagram of a home-brew board, suitable for the exercises in this guide, is shown on the following page.


pic18f2520 minimal demo board
TITLE
FILE:
PAGE 1 OF 1
REVISION: 15-Oct-2012
DRAWN BY: Peter Jacobs
U2
PIC18F2520
!MCLR/VPP/RE3
RA0/AN0
RA1/AN1
RA2/AN2/VREF-/CVREF
RA3/AN3/VREF+
RA4/T0CKI/C1OUT
RA5/AN4/!SS/HLVDIN/C2OUT
VSS
OSC1/CLKI/RA7
OSC2/CLKO/RA6
RC0/T1OSO/T13CKI
RC1/T1OSI/CCP2
RC2/CCP1
RC3/SCK/SCL
RB7/KBI3/PGD
RB6/KBI2/PGC
RB5/KBI1/PGM
RB4/KBI0/AN11
RB3/AN9/CCP2
RB2/INT2/AN8
RB1/INT1/AN10
RB0/INT0/FLT0/AN12
VDD
VSS
RC7/RX/DT
RC6/TX/CK
RC5/SDO
RC4/SDI/SDA
CONN_ICSP
!MCLR
VDD
VSS
DATA
CLK
NC
CONN_TTL_232
GND
MCU_RX
MCU_TX
CONN_RESET
CONN_I2C
+V
GND
SDA
SCL
CONN_POWER
+9 to 12V
0V
U1
IN
OUT
GND
7805
1N4148
1N4004
10k
470
330
0R
2K2
15pF
4MHz
100n
10
1u
10u
+5V
Vss

Look below for a detailed view of the home-made demo board with PIC18F2520 in place. The left photograph shows the minimal board for getting started. It is simple to make, with just header pins for the reset switch and connections to the LEDs. Two 4-pin headers are connected to the I^2C bus at the lower left and more may be added easily. The ICSP header is only needed to program FlashForth into the MCU, initially. All communication with the host PC is then via the TTL-level serial header at the lower right. The right photograph shows the same board with current-limit resistors and header pins on most of the MCU's I/O pins. This arrangement is convenient for exercises such as interfacing to the 4x3 matrix keypad (Section 8).



3 FlashForth

Forth is a word-based language, in which the data stack is made available to the programmer for temporary storage and the passing of parameters to functions. Everything is either a number or a word. Numbers are pushed onto the stack and words invoke functions. The language is simple enough to parse that full interactive Forth systems may be implemented with few (memory) resources. Forth systems may be implemented in a few kilobytes of program memory and a few hundred bytes of data memory such that it is feasible to provide the convenience of a fully interactive program development on very small microcontrollers.

The classic beginners book by Brodie [2] is available online[1], as is Pelc's more recent book [3][2]. A more detailed reference is published by Forth Inc [4]. These books are biased toward Forth running on a personal computer rather than on a microcontroller, however, they are a good place to start your reading.

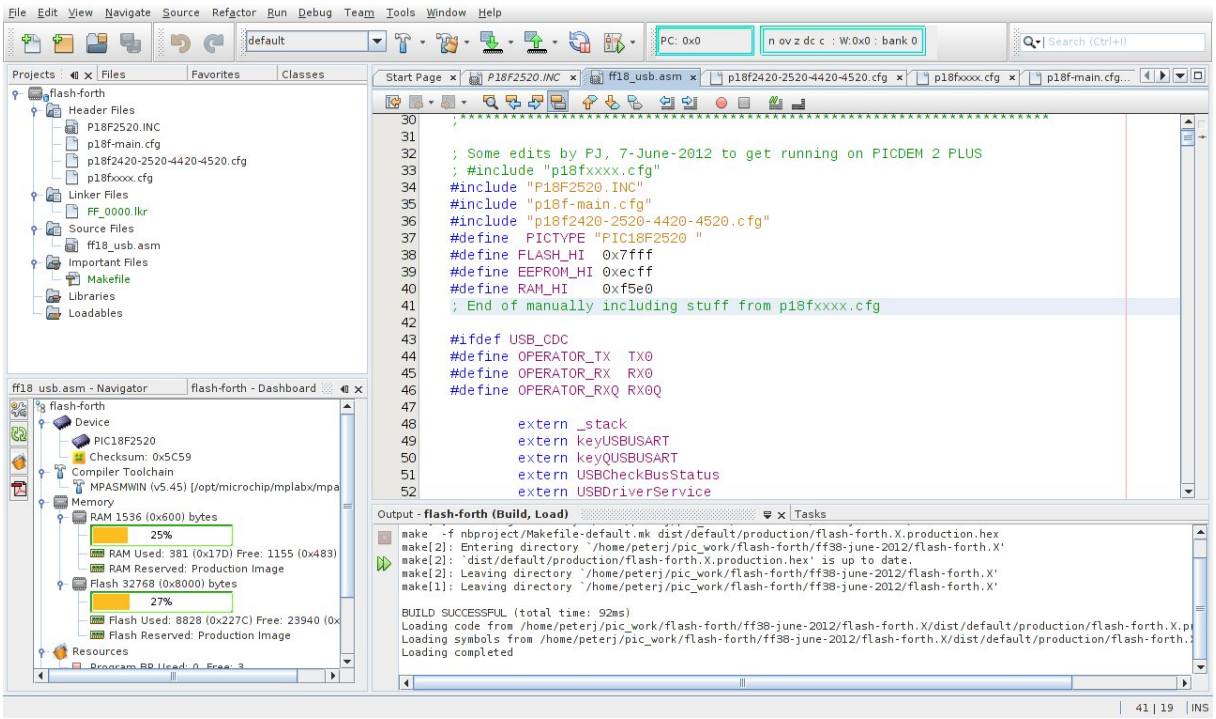
[1] `http://home.iae.nl/users/mhx/sf.html` and `http://www.forth.com/starting-forth/`
[2] `http://www.mpeforth.com/`

FlashForth for the PIC18 family of MCUs is a full interpreter and compiler that runs entirely on the MCU. It is a 16-bit Forth with a byte-addressable memory space. It does use some resources, both memory and compute cycles, but it provides such a nice interactive environment that these costs are usually returned in convenience while tinkering with your hardware. Forth programs are very compact so you will have less code to maintain in the long run. The interpreter can also be available to the end user of your instrument, possibly for making parameter adjustments or for making the hardware versatile by having a collection of application functions present simultaneously in the firmware, with the user selecting the required function as they wish.

3.1 Getting FlashForth and programming the MCU

FlashForth can be downloaded from SourceForge at the URL
`http://sourceforge.net/projects/flashforth/`
You can get a packaged release or you can clone the git repository.

For a minimal system, using the serial port as the communications channel, it is sufficient to assmeble the principal source file `ff18_usb.asm` along with the headers `pic18f-main.cfg`, `pic18f2420-2530-4420-4520.cfg`, `pic18fxxxx.cfg` and the MPLAB's include file for the processor `P18F2520.INC`. Despite the main file name alluding to USB, we build without USB capability (by leaving `USB_CDC` undefined) and use the linker script `FF_0000.lkr`. The image below shows the result of building in Microchip's MPLAB X IDE. Timer 3 (`MS_TMR3`) is used for the system ticks (milliseconds) and the watchdog timer is left enabled with a 1:256 postscale.



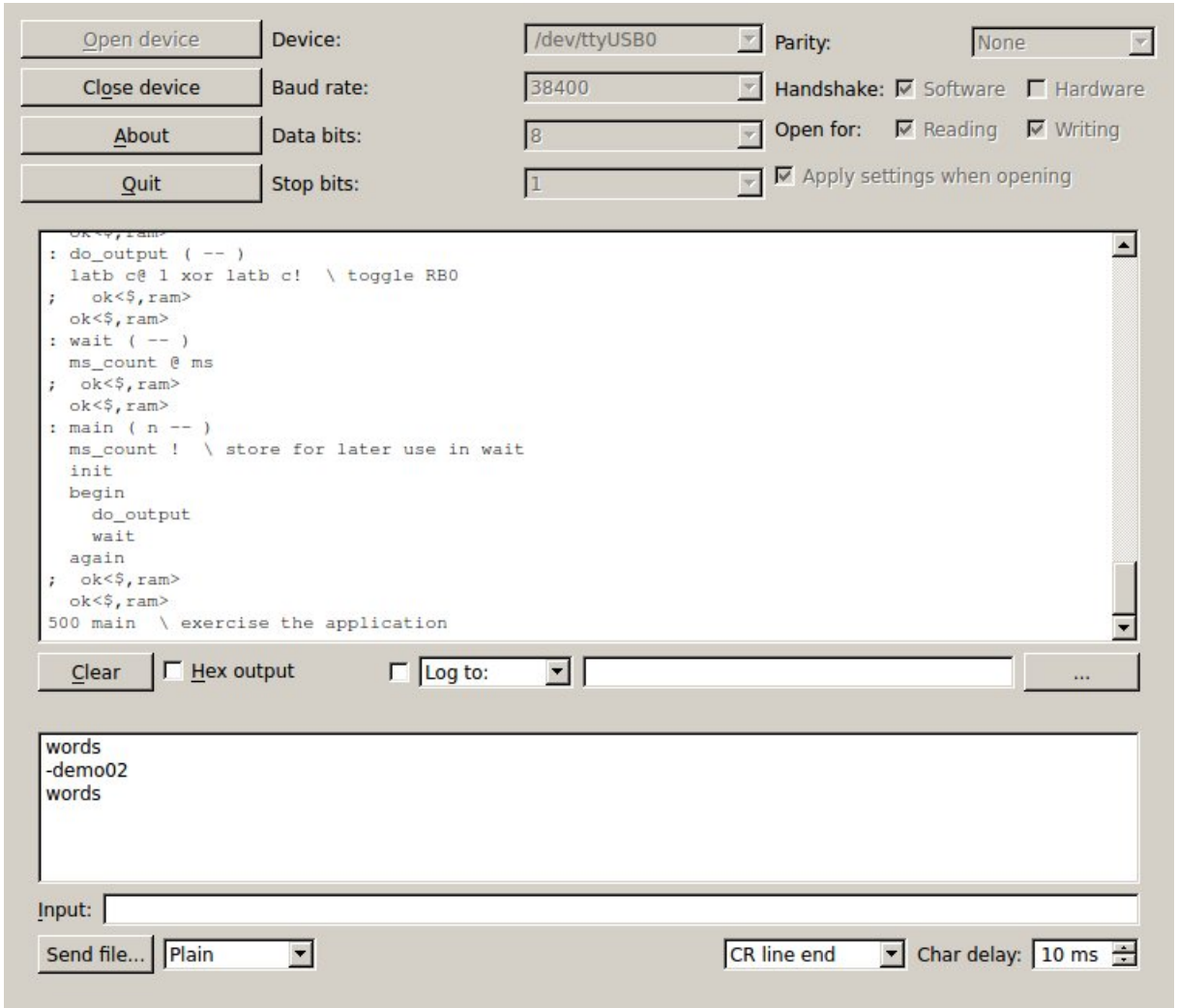
The lower left frame in MPLAB X shows the MCU resources used. With 381 bytes of SRAM used (another 1155 free) and 8828 bytes of program memory used (23940 free), For a more details on the SRAM memory map, see "The Hitchkiker's Guide to FlashForth

on PIC18 Microcontrollers". There, Mikael Nordman has provided a memory map that shows that 434 bytes are dedicated to the FlashForth system. For the PIC18F2520 MCU, FlashForth occupies only about one third of the MCU memory. The rest is available for the your application.

4 Interacting with FlashForth

Principally, interaction with the programmed MCU is via the serial port. Settings are 38400 baud 8-bit, no parity, 1 stop bit, with software flow control.

On a linux machine the `cutecom` terminal program is very convenient. It has a line-oriented input that doesn't send the text to the MCU until you press the enter key. This allows for editing of the line before committing it to the MCU and convenient recall of previous lines. The following images shows the cutecom window just afer sending the content of the `demo01.txt` file. The device name of `/dev/ttyUSB0` refers to the USB-to-serial interface that was plugged one of the PC's USB ports.



There is also a send-file capability and, importantly, the capability to set the period between characters that are sent to the serial port so as to not overwhelm the FlashForth

MCU. Although USB-to-serial interfaces usually implement software Xon-Xoff handshaking, my experience of using them with a minimal 3-wire connection (GND, RX and TX) has been variable. When sending large files, a per-character delay of 10 milliseconds had been found adequate. Of course this makes the transfer of large files very slow, however, it has the advantage that the text scrolls past slowly enough for me to watch the dialog and know how well the compilation is going.

5 Introductory examples

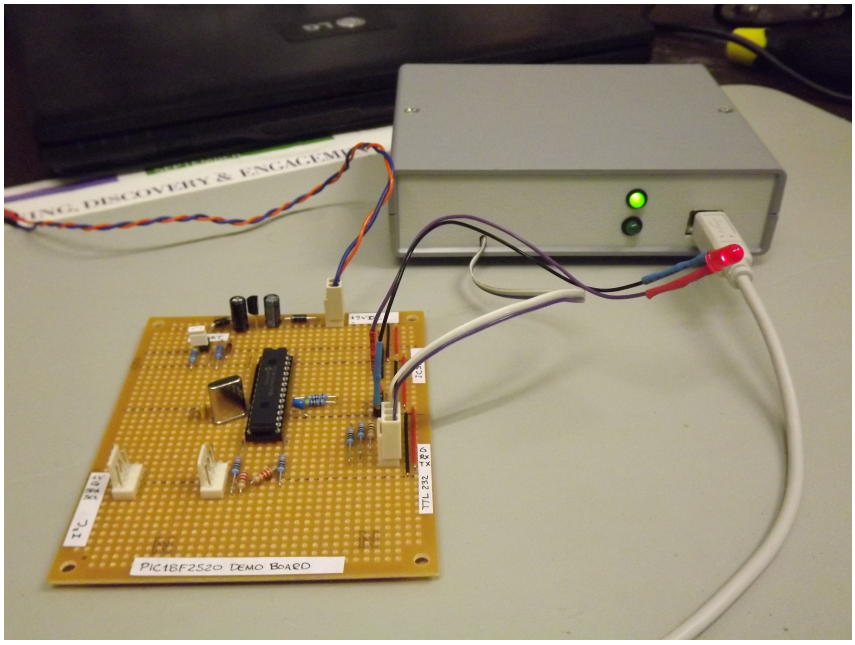
We begin with examples that demonstrate a small number of features of the MCU or of FlashForth. Our interest will primarily be in driving the various peripherals of the MCU rather than doing arithmetic or dealing with abstract data.

5.1 Hello, World: Flash a LED

The microcontroller version of the "Hello, World" program is typically a program that flashes a single LED. It makes use of a digital input-output pin via the registers that control the IO port. The datasheet [1] has a very readable introduction to the IO ports. Please read it.

```
1 marker -demo00
2 $ff8a con latb
3 $ff93 con trisb
4 : init 1 trisb mclr ; \ want RB0 as output
5 : do_output latb c@ 1 xor latb c! ; \ toggle RB0
6 : wait #500 ms ;
7 : main init begin do_output wait again ;
8 main
```

Here is the minimal demo board in action, running the flash-LED example code. The connection to the PC is via a TTL-level serial interface build from a PIC18F14K50 MCU, as per the Microchip low pin-count USB demonstration board.



Notes on this program:

- Line 1 records the state of the dictionary so that we can reset it to it's state before the code was compiled, simply by executing the word `-demo00`.
- Lines 2 and 3 define convenient names for the addresses of the file registers that control IO-port B. Note the literal hexadecimal notation with the `$` character.
- Line 4 is a colon definition for the word `init` that sets up the peripheral hardware. Here, we set pin RB0 as output. The actual command that dows the setting is `mclr`, which takes a bit-mask (00000001) and a register address (`$ff93`) and then clears the register's bits that have been set in the mask. Note the comment starting with the backslash character. Although the comment text is sent to the MCU, it is ignored. Note, also, the spaces delimiting words. That spaces after the colon and around the semicolon are important.
- Line 5 is the definition that does the work of fiddling the LED pin. We fetch the byte from the port B latch, toggle bit 0 and store the resulting byte back into the port B latch.
- Line 6 defines a word to pause for 500 milliseconds.
- Line 7 defines the "top-level" coordination word, which we have named `main`, following the C-programming convention. After initializing the relevant hardware, it unconditionally loops, doing the output operation and waiting, each pass.
- Line 8 invokes the `main` word and runs the application. Pressing the `Reset` button will kill the application and put the MCU back into a state of listening to the serial port. Typing `main`, followed by `Enter` will restart the application.

Instead of going to the bother of tinkering with the MCU IO Port, we could have taken a short-cut and used the string writing capability of Forth to write a short version that was closer the the operation of typical Hello World programs.

```
1 : greet-me ." Hello World" ;
2 greet-me
```

5.2 Set the cycle duration with a variable

We enhance the initial demonstration by making the waiting period setable. Because of the interactive FlashForth environment, the extra programming effort required is tiny. The appearance of the code, however, looks a bit different because we have laid out the colon definitions in a different style and have included more comments.

```
1 -demo01
2 marker -demo01
3 \ Flash a LED attached to pin RB0.
4
5 $ff8a con latb
6 $ff93 con trisb
7 variable ms_count \ use this for setting wait period.
8
9 : init ( -- )
10   1 trisb mclr  \ want RB0 as output
11 ;
12
13 : do_output ( -- )
14   latb c@ 1 xor latb c!  \ toggle RB0
15 ;
16
17 : wait ( -- )
18   ms_count @ ms
19 ;
20
21 : main ( n -- )
22   ms_count !  \ store for later use in wait
23   init
24   begin
25     do_output
26     wait
27   again
28 ;
29
30 #500 main  \ exercise the application
```

Notes on this program:

- If the file has been sent earlier defining the application's words, line 1 resets the state of the dictionary to forget those previous definitions. This makes it fairly convenient to have the source code open in an editing window (say, using emacs) and to simply reprogram the MCU by resending the file (with the `Send file...` button in `cutecom`).
- Line 7 defines a 16-bit variable `ms_count`.
- Line 30 leaves the wait period on the stack before invoking the `main` word.

- On each pass through the `wait` word, the 16-bit value is fetched from `ms_count` and is used to determine the duration of the pause.

5.3 Hello, World: Morse code

Staying with the minimal hardware of just a single LED, we can make a proper "Hello World" application. The following program makes use of Forth's colon definitions so that we can spell the message directly in source code and have the MCU communicate that message in Morse code.

```
-hello-world
marker -hello-world
\ Flash a LED attached to pin RB0, sending a message in Morse-code.

$ff8a con latb
$ff93 con trisb
variable ms_count \ determines the timing.

: init ( -- )
  1 trisb mclr  \ want RB0 as output
  1 latb mclr   \ initial state is off
;

: led_on 1 latb mset ;
: led_off 1 latb mclr ;
: gap ms_count @ ms ; \ pause period
: gap2 gap gap ;
: dit led_on gap led_off gap2 ;
: dah led_on gap2 led_off gap2 ;

\ Have looked up the ARRL CW list for the following letters.
: H dit dit dit dit ;
: e dit ;
: l dit dit ;
: o dah dah dah ;
: W dit dah dah ;
: r dit dah dit ;
: d dah dit dit ;

: greet ( -- )
  H e l l o  gap  W o r l d  gap2
;

: main ( n -- )
  ms_count !  \ store for later use in gap
  init
  begin
    greet
  again
;

#100 main  \ exercise the application
```

6 Read and report an analog voltage

Use of the analog-to-digital converter (ADC) is a matter of, first, reading Section 19 of the PIC18F2520 datasheet, setting the relevant configuration/control registers and then

giving it a poke when we want a measurement. Again, the interactive nature of FlashForth makes the reporting of the measured data almost trivial.

```
1  -demo02
2  marker -demo02
3  \ Read and report the analog value on RA0/AN0.
4  \ The PICDEM 2 Plus has a potentiometer attached to RA0.
5
6  $ffc4 con adresh
7  $ffc3 con adresl
8  $ffc2 con adcon0
9  $ffc1 con adcon1
10 $ffc0 con adcon2
11 $ff92 con trisa
12
13 : init ( -- )
14   1 trisa mset \ want RA0 as input
15   %00001110 adcon1 c! \ RA0 is AN0
16   %10111111 adcon2 c! \ right-justified result, long acquisition time
17   %00000001 adcon0 c! \ Power on ADC, looking at AN0
18 ;
19
20 : adc@ ( -- u )
21   %10 adcon0 mset \ Start conversion
22   begin %10 adcon0 mtst 0= until \ Wait until DONE
23   adresl @
24 ;
25
26 : wait ( -- )
27   #500 ms
28 ;
29
30 : main ( -- )
31   init
32   begin
33     adc@ u.
34     wait
35   again
36 ;
37
38 \ Exercise the application, writing digitized values periodically.
39 decimal
40 main
```

Notes on this program:

- Although not much needs to be done to set up the ADC, you really should read the ADC section of the datasheet to get the full details of this configuration.
- Lines 15 to 17 uses binary literals (with the `%` character) to show the configuration bits explicitly.
- Line 22 conditionally repeats testing of the DONE bit for the ADC.
- Line 23 fetches the full 10-bit result and leaves it on the stack for use after the `adc@` word has finished. Because of the selected configuration of the ADC peripheral, the value will be right-justified in the 16-bit cell.
- Line 33 invokes the `adc@` word and prints the numeric result.

7 Counting button presses

Example of sensing a button press, with debounce in software.

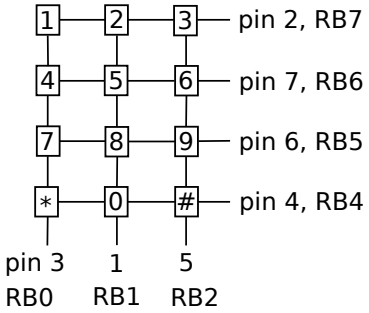
```
\ Use a push-button on RB0 to get user input.
\ This button is labelled S3 on the PICDEM2+ board.
-pb-demo
marker -pb-demo

$ff81 con portb
$ff8a con latb
$ff93 con trisb

variable count

: init ( -- )
  %01 trisb mset \ RB0 as input
  %10 trisb mclr \ RB1 as output
  %10 latb mclr
;
: RB1toggle ( -- )
  latb c@ %10 xor latb c!
;
: RB0@ ( -- c )
  portb c@ %01 and
;
: button? ( -- f )
  \ Check for button press, with software debounce.
  \ With the pull-up in place, a button press will give 0.
  RB0@ if
    0
  else
    #10 ms
    RB0@ if 0 else -1 then
  then
;

: main ( -- )
  0 count !
  init
  begin
    button? if
      RB1toggle
      count @ 1+ count !
      count @ .
      #200 ms \ allow time to release button
    then
    cwd
  again
;

main  \ exercise the application
```

Notes on this program:

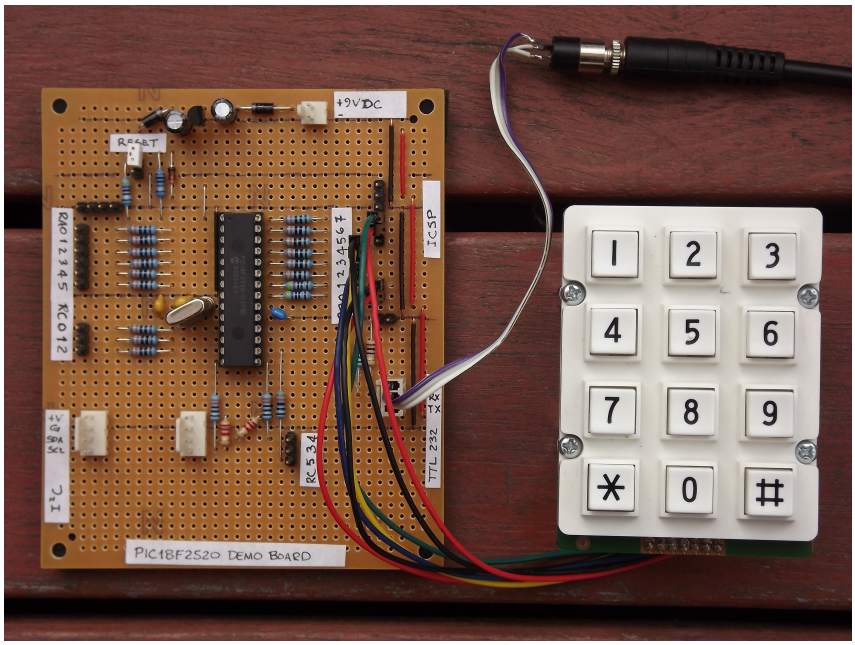
- If the pause after acknowledging the button press is too long, we may lose later button press events. This depends on how frantically we press S3.

8 Scanning a 4x3 matrix keypad

We connect a 4x3 matrix keypad to PORTB, using RB0, RB1 and RB2 to drive the columns while sensing the rows with RB4 through RB7. The schematic figure below shows the arrangement of keys and pins.



The photograph shows the demo board with jumper wires connecting the PORTB header pins to corresponding pins on the keypad. Connection to the UART is via the rather neat FTDI USB to 5 volt UART cable with the 3.5 mm connector shown in the upper right of the photograph. To minimize hardware, we have used the weak pull-ups on PORTB. Pressing a key while it's column wire is held high does nothing, however, pressing a key on a column that is held low will result in its row being pulled low.



```
-keypad
marker -keypad
\ Display key presses from a 4x3 (telephone-like) keypad

$ff81 con portb
$ff8a con latb
$ff93 con trisb
$ffc1 con adcon1
$fff1 con intcon2

: init ( -- )
  0 latb c!
  %00001111 adcon1 c! \ set as all digital I/O pins
```

```
14   %11110000 trisb c! \ RB7-4 as input, RB3-0 as output
15   %10000000 intcon2 mclr \ turn on pull-ups
16 ;
17
18 flash
19 create key_chars
20   char 1 c, char 2 c, char 3 c,
21   char 4 c, char 5 c, char 6 c,
22   char 7 c, char 8 c, char 9 c,
23   char * c, char 0 c, char # c,
24 create key_scan_bytes
25   $7e c, $7d c, $7b c,
26   $be c, $bd c, $bb c,
27   $de c, $dd c, $db c,
28   $ee c, $ed c, $eb c,
29 ram
30
31 : scan_keys ( -- c )
32   \ Return ASCII code of key that is pressed
33   #12 for
34     key_scan_bytes r@ + c@
35     dup
36     latb c!
37     portb c@
38     = if
39       \ key must be pressed to get a match
40       key_chars r@ + c@
41       rdrop
42       exit
43     then
44   next
45   0  \ no key was pressed
46 ;
47
48 : keypad@ ( -- c )
49   \ Read keypad with simple debounce.
50   \ ASCII code is left on stack.
51   \ Zero is returned for no key pressed or inconsistent scans.
52   scan_keys dup
53   #20 ms
54   scan_keys
55   = if exit else drop then
56   0 \ inconsistent scan results
57 ;
58
59 : main ( -- )
60   init
61   begin
62     keypad@
63     dup
64     0= if
65       drop  \ no key pressed
66     else
67       emit
68       #300 ms \ don't repeat key too quickly
69     then
70   again
71 ;
```

Notes on this program:

- In lines 18–29, we make use of character arrays to store (into the program memory) the the ASCII code and the scan code for each key. The scan code is made up of the 3-bit column pattern to be applied to RB2-RB0 and the resulting 4-bit row-sense pattern (RB7-RB4) expected for the particular key if it is pressed. RB3 is

maintained high (and is of no consequence) for this 3-column keypad, however, it would be used for a 4x4 keypad.

- Lines 33 and 44 make use of the for–next control construct to work through the set of 12 scan codes
- We should go further by making use a state-machine and also keeping track of the last key pressed.

9 Using I²C to get temperature measurements

Using the MSSP peripheral in master mode, one may talk to the TC74A5 temperature measurement chip on the PICDEM 2 PLUS and report sensor temperature.

```
1  \ Read temperature from TC74 on PICDEM2+ board.
2  \ Modelled on Mikael Nordman's i2c_tcn75.txt.
3  \ This program requires i2c_base.txt to be previously loaded.
4  -read-tc74
5  marker -read-tc74
6
7  %1001101 con addr-tc74  \ default 7-bit address for TC74
8
9  : add-read-bit ( 7-bit-c -- 8-bit-c )
10   \ Make 8-bit i2c address with bit 0 set.
11   1 lshift 1 or
12 ;
13 : add-write-bit ( 7-bit-c -- 8-bit-c )
14   \ Make 8-bit i2c address with bit 0 clear.
15   1 lshift 1 invert and
16 ;
17 : sign-extend ( c -- n )
18   \ If the TC74 has returned a negative 8-bit value,
19   \ we need to sign extend to 16-bits with ones.
20   dup $7f > if $ff80 or then
21 ;
22 : init-tc74 ( -- )
23   \ Selects the temperature register for subsequent reads.
24   addr-tc74 add-write-bit i2cws 0 i2c! spen
25 ;
26 : degrees@ ( -- n )
27   \ Wake the TC74 and receive its register value.
28   addr-tc74 add-read-bit i2cws i2c@nak sign-extend
29 ;
30 : main ( -- )
31   i2cinit
32   init-tc74
33   begin
34     degrees@ .
35     #1000 ms
36   again
37 ;
38
39 \ Now, report temperature in degrees C
40 \ while we warm up the TC74 chip with our fingers...
41 decimal main
```

With a Saleae Logic Analyser connected to the pins of the TC74A5, we can see the I²C signals as a result of calling the `init-tc74` word.

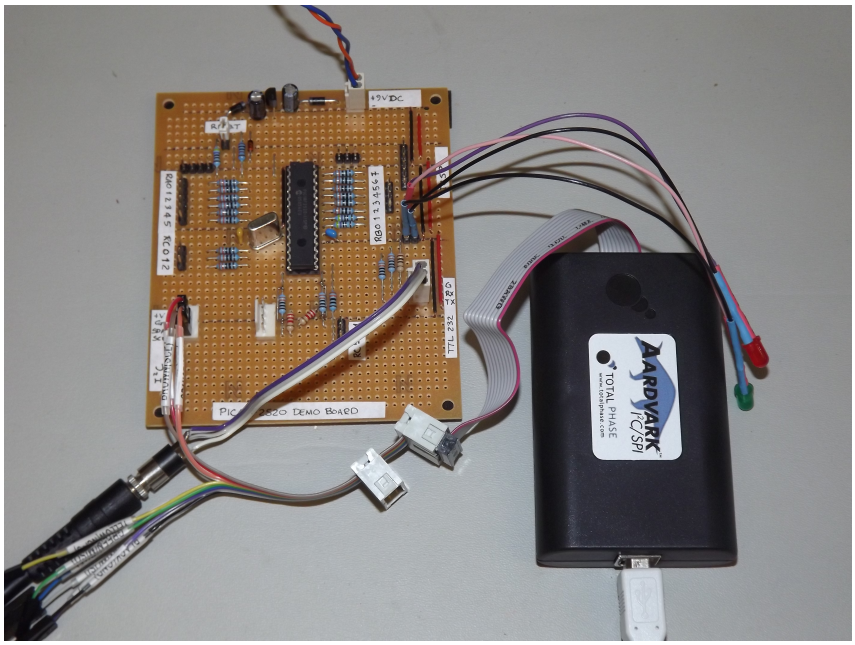
A little later on, the `degrees@` word is invoked. The returned binary value of `0b00010101` corresponds to the very pleasant 21°C that exists in the back shed as this text is being written.

Notes on this program:

- This builds upon the `i2c_base` and `asm` words supplied with FlashForth.

10 An I²C slave example

The MSSP in the PIC18F2520 can also be used in slave mode. Here, the FlashForth demo board is presented as an I²C slave device to an Aardvark serial interface, acting as master. The UART communication is provided by a Future Technology Devices International USB TTL-serial cable.



The core of the program is the `i2c_service` word which is invoked each time a serial-port event is flagged by the SSPIF bit in the PIR1 flag register. This word is an implementation of the state look-up approach detailed in the Microchip Application Note AN734 [5]. The rest of the program is there to provide (somewhat) interesting data for the I²C master to read and to do something (light a LED) when the master writes suitable data to the slave.

```
-i2c-slave
marker -i2c-slave
\ Make the FlashForth demo board into an I2C slave.
\ An I2C master can read and write to a buffer here,
\ the least-significant bit of the first byte controls
\ the LED attached to pin RB0.

$ff81 con portb
$ff82 con portc
$ff8a con latb
$ff93 con trisb
$ff94 con trisc

: led_on ( -- )
  %00000001 latb mset
;
: led_off ( -- )
  %00000001 latb mclr
;
: err_led_on ( -- )
  %00000010 latb mset
;
: err_led_off ( -- )
  %00000010 latb mclr
;

\ Establish a couple of buffers in RAM, together with index variables.
ram
8 con buflen
\ Receive buffer for incoming I2C data.
create rbuf buflen allot
variable rindx
: init_rbuf ( -- )
  rbuf buflen erase
  0 rindx !
;
: incr_rindx ( -- ) \ increment with wrap-around
  rindx @ 1 +
  dup buflen = if drop 0 then
  rindx !
;
: save_to_rbuf ( c -- )
  rbuf rindx @ + c!
  incr_rindx
;

\ Send buffer with something interesting for the I2C master to read.
create sbuf buflen allot
variable sindx
: incr_sindx ( -- ) \ increment with wrap-around
  sindx @ 1 +
  dup buflen = if drop 0 then
  sindx !
;
: init_sbuf ( -- ) \ fill with counting integers, for interest
  buflen
  for
    r@ 1+
    sbuf r@ + c!
  next
  0 sindx !
;

\ I2C-related definitions and code
$ffc5 con sspcon2
$ffc6 con sspcon1
$ffc7 con sspstat
$ffc8 con sspadd
$ffc9 con sspbuf
$ff9e con pir1
```

```
71
72 \ PIR1 bits
73 %00001000 con sspif
74
75 \ SSPSTAT bits
76 %00000001 con bf
77 %00000100 con r_nw
78 %00001000 con start_bit
79 %00010000 con stop_bit
80 %00100000 con d_na
81 %01000000 con cke
82 %10000000 con smp
83
84 d_na start_bit or r_nw or bf or con stat_mask
85
86 \ SSPCON1 bits
87 %00010000 con ckp
88 %00100000 con sspen
89 %01000000 con sspov
90 %10000000 con wcol
91
92 \ SSPCON2 bits
93 %00000001 con sen
94
95 : i2c_init ( -- )
96   %00011000 trisc mset \ RC3==SCL RC4==SDA
97   %00000110 sspcon1 c! \ Slave mode with 7-bit address
98   sen sspcon2 mset \ Clock stretching enabled
99   smp sspstat mset \ Slew-rate disabled
100  $52 1 lshift sspadd c! \ Slave address
101  sspen sspcon1 mset \ Enable MSSP peripheral
102 ;
103
104 : release_clock ( -- )
105   ckp sspcon1 mset
106 ;
107
108 : i2c_service ( -- )
109   \ Check the state of the I2C peripheral and react.
110   \ See App Note 734 for an explanation of the 5 states.
111   \
112   \ State 1: i2c write operation, last byte was address.
113   \ D_nA=0, S=1, R_nW=0, BF=1
114   sspstat c@ stat_mask and %00001001 =
115   if
116     sspbuf @ drop
117     init_rbuf
118     release_clock
119     exit
120   then
121   \ State 2: i2c write operation, last byte was data.
122   \ D_nA=1, S=1, R_nW=0, BF=1
123   sspstat c@ stat_mask and %00101001 =
124   if
125     sspbuf c@ save_to_rbuf
126     release_clock
127     exit
128   then
129   \ State 3: i2c read operation, last byte was address.
130   \ D_nA=0, S=1, R_nW=1
131   sspstat c@ %00101100 and %00001100 =
132   if
133     sspbuf c@ drop
134     0 sindx !
135     wcol sspcon1 mclr
136     sbuf sindx @ + c@ sspbuf c!
137     release_clock
138     incr_sindx
139     exit
140   then
141   \ State 4: i2c read operation, last byte was outgoing data.
```

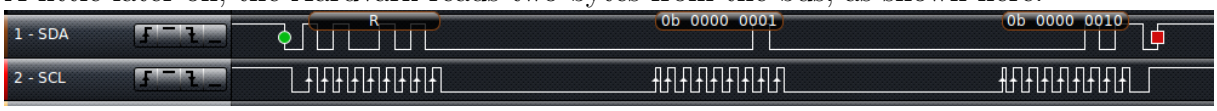
```
142   \ D_nA=1, S=1, R_nW=1, BF=0
143   sspstat c@ stat_mask and %00101100 =
144   ckp sspcon1 mtst 0=
145   and
146   if
147     wcol sspcon1 mclr
148     sbuf sindx @ + c@ sspbuf c!
149     release_clock
150     incr_sindx
151     exit
152   then
153   \ State 5: master NACK, slave i2c logic reset.
154   \ From AN734: D_nA=1, S=1, BF=0, CKP=1, however,
155   \ we use just D_nA=1 and CKP=1, ignoring START bit.
156   \ This is because master may have already asserted STOP
157   \ before we service the final NACK on a read operation.
158   d_na sspstat mtst 0 > ckp sspcon1 mtst 0 > and
159   stop_bit sspstat mtst or
160   if
161     exit \ Nothing needs to be done.
162   then
163   \ We shouldn't arrive here...
164   err_led_on
165   cr ." Error "
166   ." sspstat " sspstat c@ u.
167   ." sspcon1 " sspcon1 c@ u.
168   ." sspcon2 " sspcon2 c@ u.
169   cr
170   begin again \ Hang around until watch-dog resets MCU.
171 ;
172
173
174 : init ( -- )
175   %00000011 trisb mclr  \ want RB0,RB1 as output pins
176   init_rbuf
177   init_sbuf
178   i2c_init
179   led_on err_led_on #200 ms led_off err_led_off
180 ;
181
182 : main ( n -- )
183   cr ." Start I2C slave "
184   init
185   begin
186     sspif pir1 mtst
187     if
188       sspif pir1 mclr
189       i2c_service
190     then
191     rbuf c@ %00000001 and
192     if led_on else led_off then
193     cwd
194   again
195 ;
196
197 \ ' main is turnkey
```

With a Saleae Logic Analyser connected, we can see the I²C signals as a result of writing the byte `0x01` to turn on the LED. The following figure shows the data and clock signals from the time that the master asserts the START condition (green circle) until it asserts the STOP condition (as indicated by the red square).

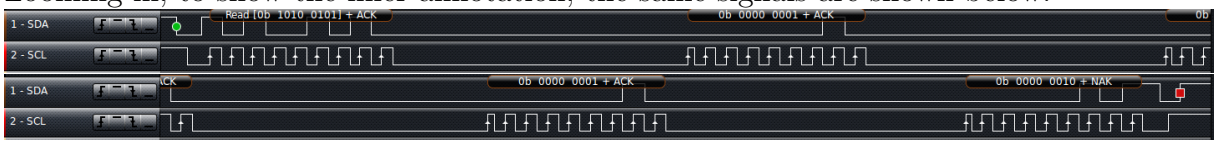
The clock frequency is 100kHz and there is a 138 μs gap between the ninth clock pulse of

the address byte and the start of the pulses for the data byte. This gives an indication of the time needed to service each SSPIF event.

A little later on, the Aardvark reads two bytes from the bus, as shown here.



Zooming in, to show the finer annotation, the same signals are shown below.



Again, the inter-byte gap is 138 μs resulting in about 200 μs needed to transfer each byte. This effective speed of 5 kbytes/s should be usable for many applications, since the I²C bus is typically used for low speed data transfer.

Notes on this program:

- Need to load `core.txt` before the source code of the `i2c-slave.txt`.
- Slave examples found in documentation on the Web usually have the service function written in the context of an interrupt service routine. The MSSP can be serviced quite nicely without resorting to the use of interrupts, however, you still have to check and clear the SSPIF bit for each event.
- The implementation of the test for State 5 (Master NACK) is slightly different to that described in AN734 because it was found that the master would assert an I²C bus stop after the final NACK of a read operation but before the MCU could service the SSPIF event. This would mean that STOP was the most recent bus condition seen by the MSSP and the START and STOP bits set to reflect this. In the figures shown above, there is only about 12 μs between the ninth clock pulse for the second read data byte and the Aardvark master asserting the STOP condition on the bus. This period is very much shorter than the (approx.) 140 μs period needed by the slave firmware to service the associated SSPIF event.

11 Speed of operation

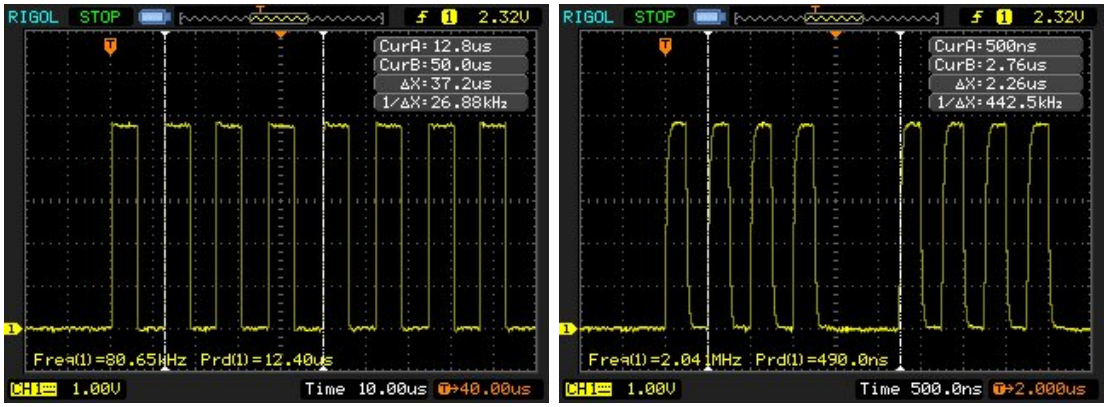
All of this nice interaction and convenience has some costs. One cost is the number of MCU instruction cycles needed to process the Forth words. To visualize this cost, the following program defines a word which toggles an IO pin using the (high-level) FlashForth words and an alternative word uses assembler instructions to achieve an equivalent effect.

```
1  -speed-test
2  marker -speed-test
3  \ Waggle RB1 as quickly as we can, in both high- and low-level code.
4  \ Before sending this file, we should send asm.txt so that we have
5  \ the clrwdt, word available.
6
7  $ff8a con latb
8  $ff93 con trisb
9
10 : initRB1
11   %10 trisb mclr \ RB1 as output
12   %10 latb mclr \ initially known state
13 ;
14
15 \ high-level bit fiddling, presumably slow
16 : blink-forth ( -- )
17   initRB1
18   begin
19     %10 latb c! 0 latb c! \ one cycle, on and off
20     %10 latb c! 0 latb c!
21     %10 latb c! 0 latb c!
22     %10 latb c! 0 latb c!
23     cwd \ We have to kick the watch dog ourselves.
24   again
25 ;
26
27 \ low-level bit fiddling, via assembler
28 : blink-asm ( -- )
29   initRB1
30   [
31   begin,
32     latb 1 a, bsf,  latb 1 a, bcf, \ one cycle, on and off
33     latb 1 a, bsf,  latb 1 a, bcf,
34     latb 1 a, bsf,  latb 1 a, bcf,
35     latb 1 a, bsf,  latb 1 a, bcf,
36     clrwdt, \ kick the watch dog
37   again,
38   ]
39 ;
```

Notes on this program:

- We have had to worry about clearing the watch-dog timer. In the early examples, the FlashForth interpreter was passing through the pause state often enough to keep the watch-dog happy. The words in this example give the FlashForth interpreter no time to pause so we are responsible for clearing the watch-dog timer explicitly.
- In the source code config file for the specific MCU, the watch-dog timer postscale is set to 256. With an 8 MHz internal RC frequency, this leads to a default timeout period of a little over 4 milliseconds ($0.125\,\mu\text{s} \times 128 \times 256$).
- The MCU on the PICDEM 2 PLUS board is running off the external 4 MHz crystal and has the 4× PLL enabled. This leads to an instruction cycle period of 250 ns.

- The screen image on the left shows the output signal for running the high-level Forth code while the image on the right uses the assembler words.



- For the pure Forth code, one on+off cycle of the LED executes in 6 words and is seen (in the oscilloscope record) to require about 51 instruction cycles. So, on average, these threaded Forth words, are executed in about 8 MCU instructions. Note that this overhead includes the cost of using 16-bit cells for the data. Extra machine instructions are used to handle the upper bytes. In other applications, where we actually want to handle 16-bit data, this will no longer be a penalty.

- The assembler version has no overhead and the cycle time for the MCU instructions defines the period of the output signal. One on-off cycle requires 2 instructions so we see a short 500 μs period. This is fast enough that the capacitive loading on the output pin is noticeable in the oscilloscope trace. Also, the time required for the machine instructions to clear the watch-dog timer and the instruction jump back to the start of the loop now shows up clearly in the oscilloscope record.

References

[1] Microchip Technology Inc. PIC18(L)F2420/2520/4420/4520 data sheet: 28/40/44-pin enhanced flash microcontrollers with 10-bit A/D and nanowatt technology. Datasheet DS39631E, Microchip Technology Inc., www.microchip.com, 2008.

[2] L Brodie and Forth Inc. *Starting Forth: An introduction to the Forth Language and operating system for beginners and professionals, 2nd Ed.* Prentice Hall, Englewood Cliffs, New Jersey, 1987.

[3] S. Pelc. *Programming Forth.* Microprocessor Engineering Limited, 2011.

[4] E. K. Conklin and E. D. Rather. *Forth Programmer's Handbook, 3rd Ed.* Forth Inc., California, 2007.

[5] S. Bowling and N. Raj. Using the PIC devices SSP and MSSP modules for slave I2C communication. Application Note AN734, Microchip Technology Inc., 2008.