A Tutorial Guide to Programming PIC18, PIC24 and ATmega Microcontrollers with FlashForth. 2016 Revision

Mechanical Engineering Report 2016/01
P. A. Jacobs
School of Mechanical and Mining Engineering
The University of Queensland.

January 27, 2016

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 the 28-pin PIC18F26K22 microcontroller and its 40-pin PIC18F46K22 sibling in a simple hardware environment. A number of example programs, in the Forth language, are provided to illustrate the use of some of each microcontroller's peripheral devices. The examples cover the very simple "flash a LED" exercise through to driving a character-based LCD via its 4-bit parallel interface. The set-up and use of FlashForth 5 on the PIC24FV32KA302 and AVR ATmega328P microcontrollers is also covered.

Contents

1 A selection of microcontrollers 4

2 Development boards 7
2.1 PIC18 family boards 7
2.2 AVR and PIC24 boards 10

3 FlashForth 14
3.1 Getting FlashForth and programming the MCU 14
3.2 Building for the PIC18F26K22 or PIC18F46K22 15
3.3 Building for the PIC24FV32KA302 17
3.4 Building for the ATmega328P 17

4 Interacting with FlashForth 18

5 Introductory examples 20
5.1 Hello, World: Flash a LED on the PIC18FX6K22 20
5.2 Flash a LED on the PIC24FV32KA302 21
5.3 Flash a LED on the ATmega328P 22
5.4 Set the cycle duration with a variable (PIC18FX6K22) 23
5.5 Hello, World: Morse code 24

6 Read and report an analog voltage 25
6.1 PIC18FX6K22 25
6.2 PIC24FV32KA30X 26

7 Counting button presses 28

8 Counting button presses via interrupts 30

9 Scanning a 4x3 matrix keypad 32

10 Base words for an I^2C master 34
10.1 PIC18FX6K22 34
10.2 PIC24FV32KA30X 36
10.3 ATmega328P 38
10.4 Notes on using the words 40
10.5 Detecting I^2C devices 41

11 Using I^2C to get temperature measurements 42

12 Making high-resolution voltage measurements 43

13 An I^2C slave example 45

14 Speed of operation – bit banging **49**
14.1 PIC18F26K22 49
14.2 PIC24FV32KA302 51
14.3 ATmega328P 53

15 Driving an Hitachi-44780 LCD controller **56**

A Using other terminal programs on Linux **60**

1 A selection of microcontrollers

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 an 8-bit Microchip PIC18F26K22 or PIC18F46K22 microcontroller, a 16-bit Microchip PIC24FV32KA302 microcontroller and an 8-bit Atmel ATmega328P microcontroller, all programmed with the FlashForth version 5 interpreter [1].

Within each family of Microchip or Atmel microcontrollers, the individual 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 microcontrollers that have:

- 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 PIC18F26K22 and PIC18F46K22 MCUs on the following pages.
- a 28-pin narrow or 40-pin DIL package, which is convenient for prototyping and has enough I/O pins to play without needing very careful planning.
- an ability to work as 3.3V or 5V systems.
- a pinout as shown at the start of the datasheets (books) [2, 3, 4]. You will be reading the pages of these books over and over but we include the following couple of pages from the PIC18F22K26/PIC18F46K22 datasheet to give an overview.
- an internal arrangement that is built around an 8-bit or 16-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 FlashForth interpreter handle most of the details, however, memory layout, especially the I/O memory layout is important for us as programmers. The peripheral devices, which are used to inferface with the real world, are controlled and accessed via registers in the data-memory space.

PIC18(L)F2X/4XK22

28/40/44-Pin, Low-Power, High-Performance Microcontrollers with XLP Technology

High-Performance RISC CPU:

- C Compiler Optimized Architecture:
  - Optional extended instruction set designed to optimize re-entrant code
- Up to 1024 Bytes Data EEPROM
- Up to 64 Kbytes Linear Program Memory Addressing
- Up to 3896 Bytes Linear Data Memory Addressing
- Up to 16 MIPS Operation
- 16-bit Wide Instructions, 8-bit Wide Data Path
- Priority Levels for Interrupts
- 31-Level, Software Accessible Hardware Stack
- 8 x 8 Single-Cycle Hardware Multiplier

Flexible Oscillator Structure:

- Precision 16 MHz Internal Oscillator Block:
  - Factory calibrated to ± 1%
  - Selectable frequencies, 31 kHz to 16 MHz
  - 64 MHz performance available using PLL – no external components required
- Four Crystal modes up to 64 MHz
- Two External Clock modes up to 64 MHz
- 4X Phase Lock Loop (PLL)
- Secondary Oscillator using Timer1 @ 32 kHz
- Fail-Safe Clock Monitor:
  - Allows for safe shutdown if peripheral clock stops
  - Two-Speed Oscillator Start-up

Analog Features:

- Analog-to-Digital Converter (ADC) module:
  - 10-bit resolution, up to 30 external channels
  - Auto-acquisition capability
  - Conversion available during Sleep
  - Fixed Voltage Reference (FVR) channel
  - Independent input multiplexing
- Analog Comparator module:
  - Two rail-to-rail analog comparators
  - Independent input multiplexing
- Digital-to-Analog Converter (DAC) module:
  - Fixed Voltage Reference (FVR) with 1.024V, 2.048V and 4.096V output levels
  - 5-bit rail-to-rail resistive DAC with positive and negative reference selection
- Charge Time Measurement Unit (CTMU) module:
  - Supports capacitive touch sensing for touch screens and capacitive switches

Extreme Low-Power Management PIC18(L)F2X/4XK22 with XLP:

- Sleep mode: 20 nA, typical
- Watchdog Timer: 300 nA, typical
- Timer1 Oscillator: 800 nA @ 32 kHz
- Peripheral Module Disable

Special Microcontroller Features:

- 2.3V to 5.5V Operation – PIC18FXXK22 devices
- 1.8V to 3.6V Operation – PIC18LFXXK22 devices
- Self-Programmable under Software Control
- High/Low-Voltage Detection (HLVD) module:
  - Programmable 16-Level
  - Interrupt on High/Low-Voltage Detection
- Programmable Brown-out Reset (BOR):
  - With software enable option
  - Configurable shutdown in Sleep
- Extended Watchdog Timer (WDT):
  - Programmable period from 4 ms to 131s
- In-Circuit Serial Programming™ (ICSP™):
  - Single-Supply 3V
- In-Circuit Debug (ICD)

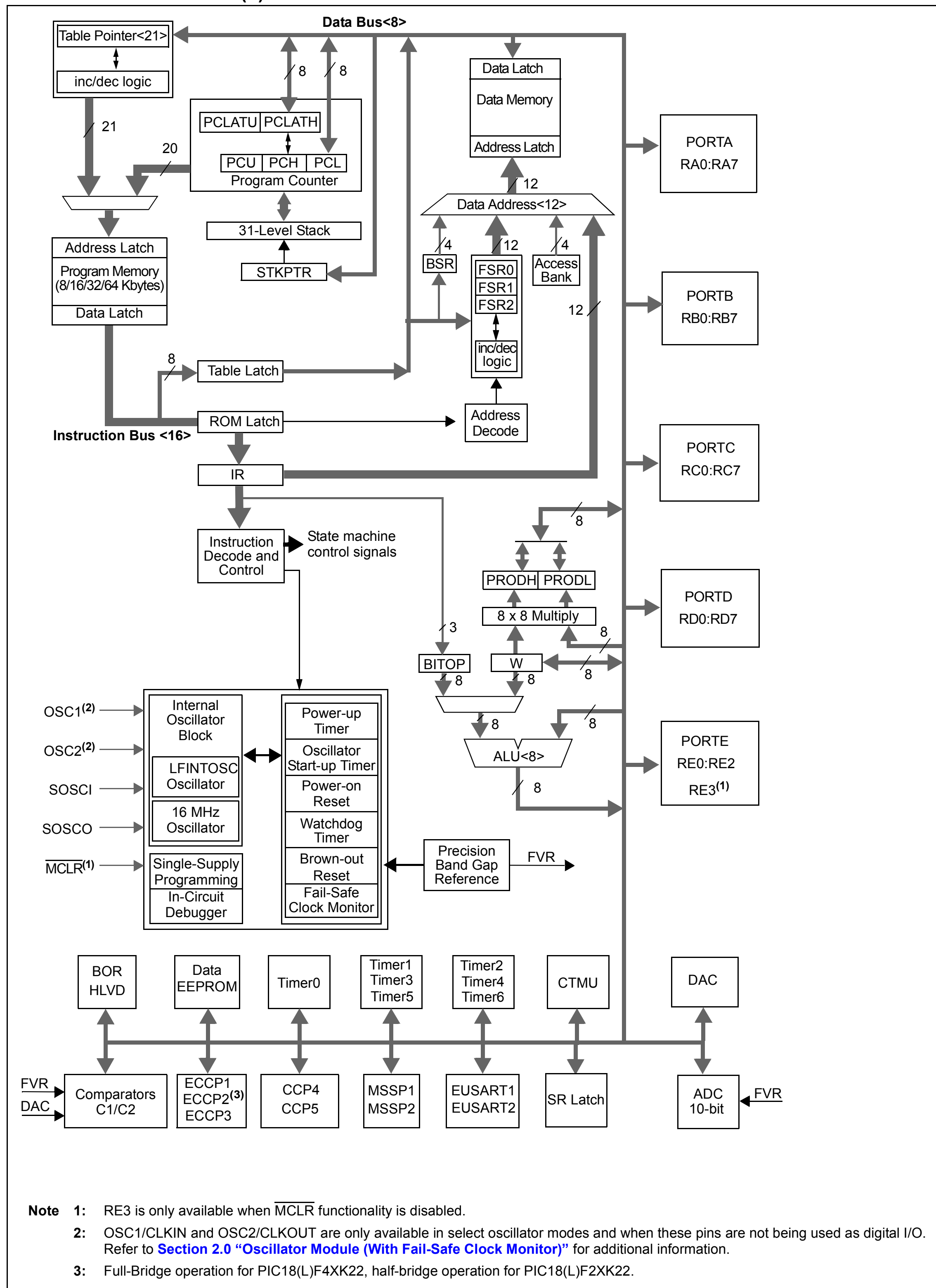
Peripheral Highlights:

- Up to 35 I/O Pins plus 1 Input-Only Pin:
  - High-Current Sink/Source 25 mA/25 mA
  - Three programmable external interrupts
  - Four programmable interrupt-on-change
  - Nine programmable weak pull-ups
  - Programmable slew rate
- SR Latch:
  - Multiple Set/Reset input options
- Two Capture/Compare/PWM (CCP) modules
- Three Enhanced CCP (ECCP) modules:
  - One, two or four PWM outputs
  - Selectable polarity
  - Programmable dead time
  - Auto-Shutdown and Auto-Restart
  - PWM steering
- Two Master Synchronous Serial Port (MSSP) modules:
  - 3-wire SPI (supports all 4 modes)
  - I^2C™ Master and Slave modes with address mask

© 2010-2012 Microchip Technology Inc.

PIC18(L)F2X/4XK22

FIGURE 1-1: PIC18(L)F2X/4XK22 FAMILY BLOCK DIAGRAM



Note 1: RE3 is only available when MCLR functionality is disabled.

2: OSC1/CLKIN and OSC2/CLKOUT 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 Module (With Fail-Safe Clock Monitor)"** for additional information.

3: Full-Bridge operation for PIC18(L)F4XK22, half-bridge operation for PIC18(L)F2XK22.

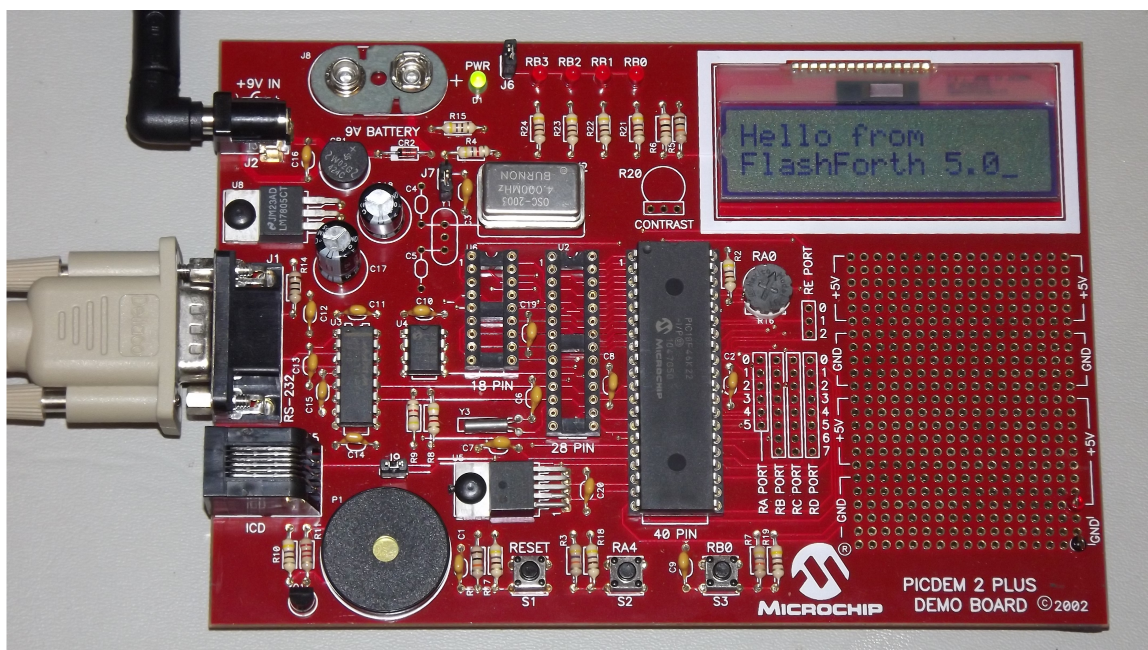
© 2010-2012 Microchip Technology Inc.

2 Development boards

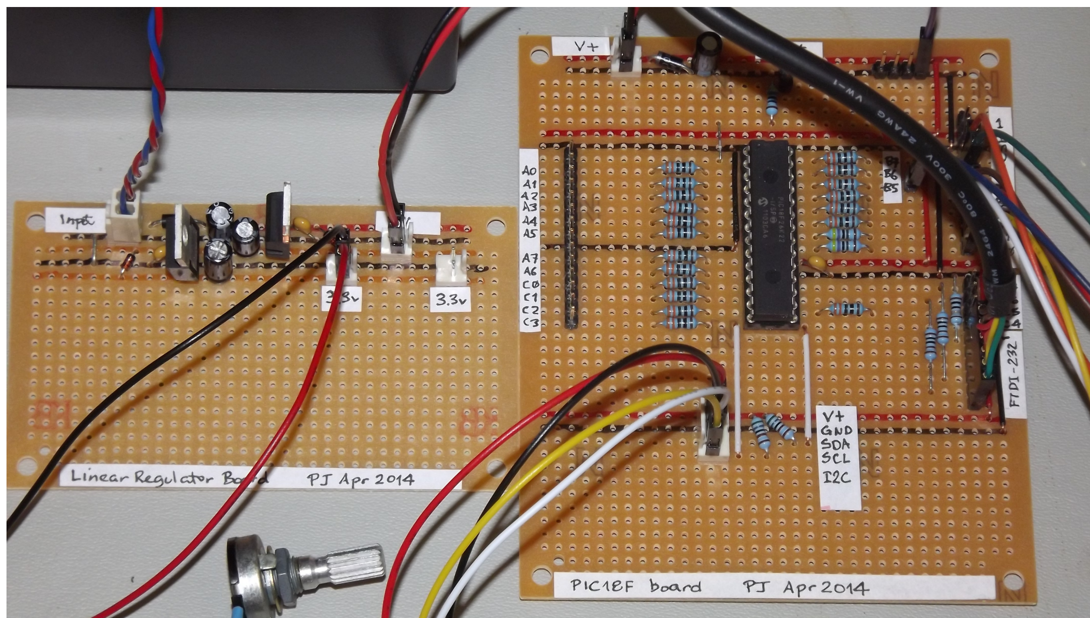
This tutorial is based around simple support hardware for each of the microcontrollers. If you don't want to do your own soldering, there are easy-to-buy demonstration boards available as a convenient way to get your hardware up and going. If you are a student of mechatroncis, however, you must eventually design and build your own hardware. The strip-board versions are aimed at you.

2.1 PIC18 family boards

Here is a picture of PICDEM 2 PLUS with PIC18F46K22-I/P in the 40-pin socket (U1) and running the LCD, as described in Section 15. 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 I²C 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 (labelled ICD in the lower left of the photograph) with a Microchip MPLAB-ICD3, PICkit3, or similar device programmer.

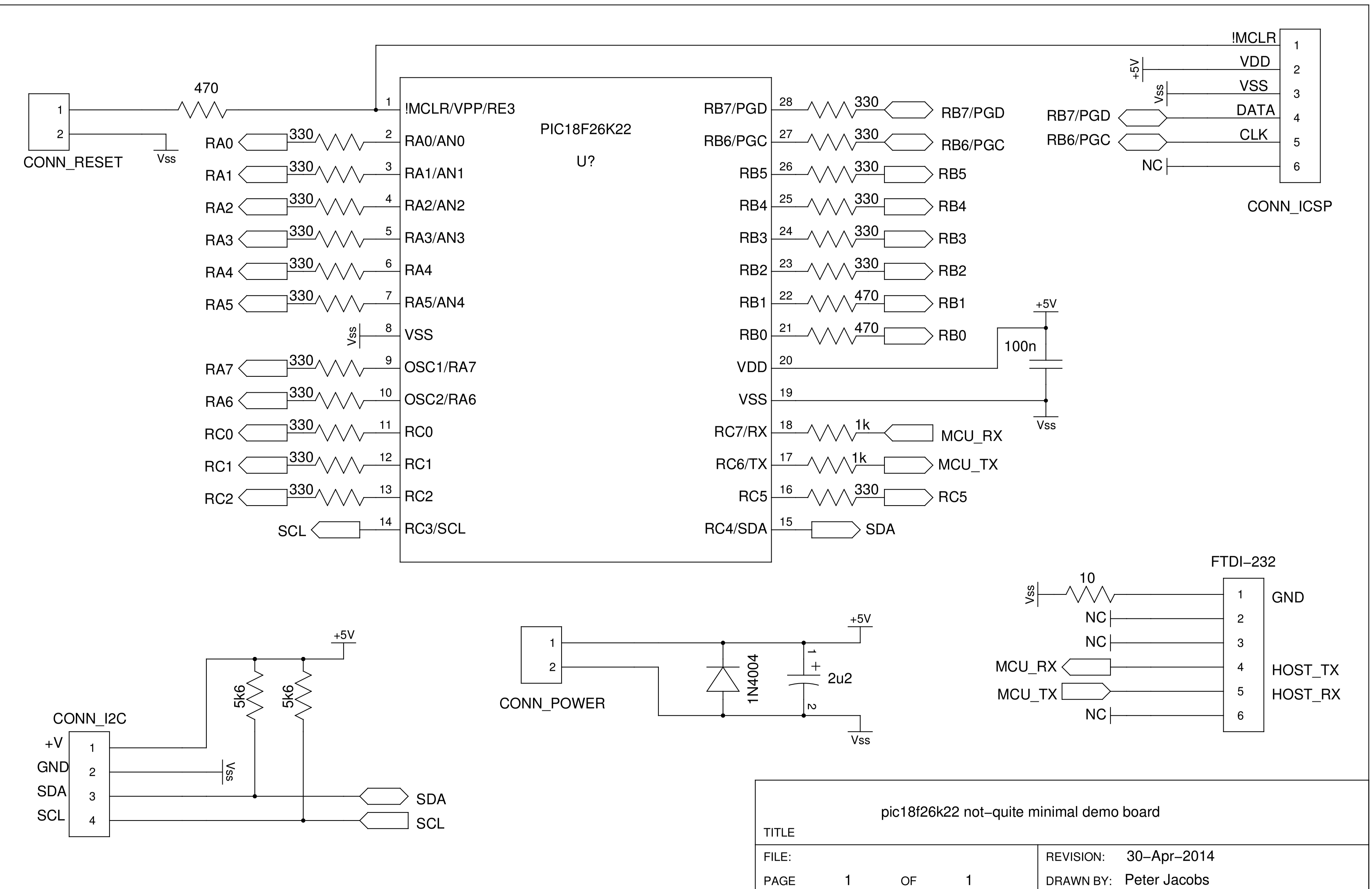


If you want a homebrew system, you can build a minimal system on strip-board that works well. One of the nice things about such a strip-board construction is that you can easily continue construction of your bespoke project on the board and, with careful construction, your prototype can provide years of reliable service.



Here is a detailed view of the home-made demo board with PIC18F26K22 in place. This board is suitable for the exercises in this guide. A separate regulator board is to the left and a current-limited supply provides the input power. The board is simple to make by hand, with header pins for the reset switch and connections to the LEDs. The 4-pin header in the foreground provides an I^2C connection. 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 (labelled FTDI-232) at the right. Beyond the minimum required to get the microcontroller to function, we have current-limiting 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 9).

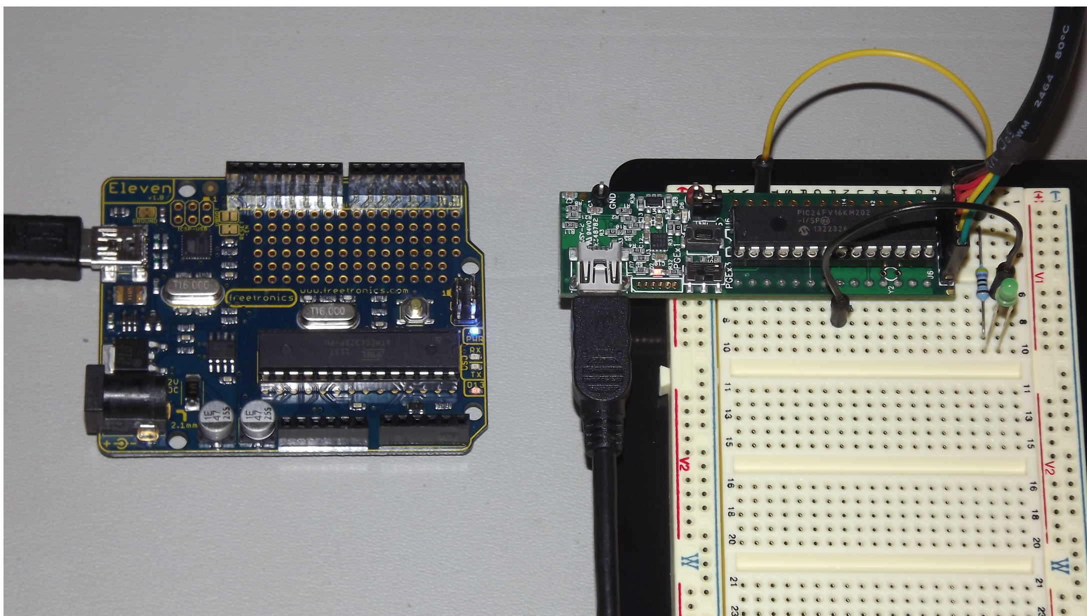
The schematic diagram of this home-brew board is shown on the following page. Note that there is no crystal oscillator on the board; the internal oscillator is sufficiently accurate for asynchronous serial port communication. Note, also, the 1k resistors in the TX and RX nets. These limit the current going through the microcontroller pin-protection diodes in the situation where the microcontroller board is unpowered and the FTDI-232 cable is still plugged in to your PC. This will happen at some point and, without the current-limiting resistors, the FTDI cable will power the microcontroller, probably poorly.


CONN_RESET
470
Vss
!MCLR/VPP/RE3
RA0/AN0
RA1/AN1
RA2/AN2
RA3/AN3
RA4
RA5/AN4
VSS
OSC1/RA7
OSC2/RA6
RC0
RC1
RC2
RC3/SCL
PIC18F26K22
U?
RB7/PGD
RB6/PGC
RB5
RB4
RB3
RB2
RB1
RB0
VDD
VSS
RC7/RX
RC6/TX
RC5
RC4/SDA
RA0
RA1
RA2
RA3
RA4
RA5
RA7
RA6
RC0
RC1
RC2
SCL
SDA
330
470
1k
100n
+5V
MCU_RX
MCU_TX
!MCLR
VDD
VSS
DATA
CLK
NC
CONN_ICSP
FTDI–232
10
GND
HOST_TX
HOST_RX
CONN_POWER
1N4004
2u2
CONN_I2C
+V
GND
SDA
SCL
5k6
pic18f26k22 not–quite minimal demo board
TITLE
FILE:
PAGE 1 OF 1
REVISION: 30–Apr–2014
DRAWN BY: Peter Jacobs

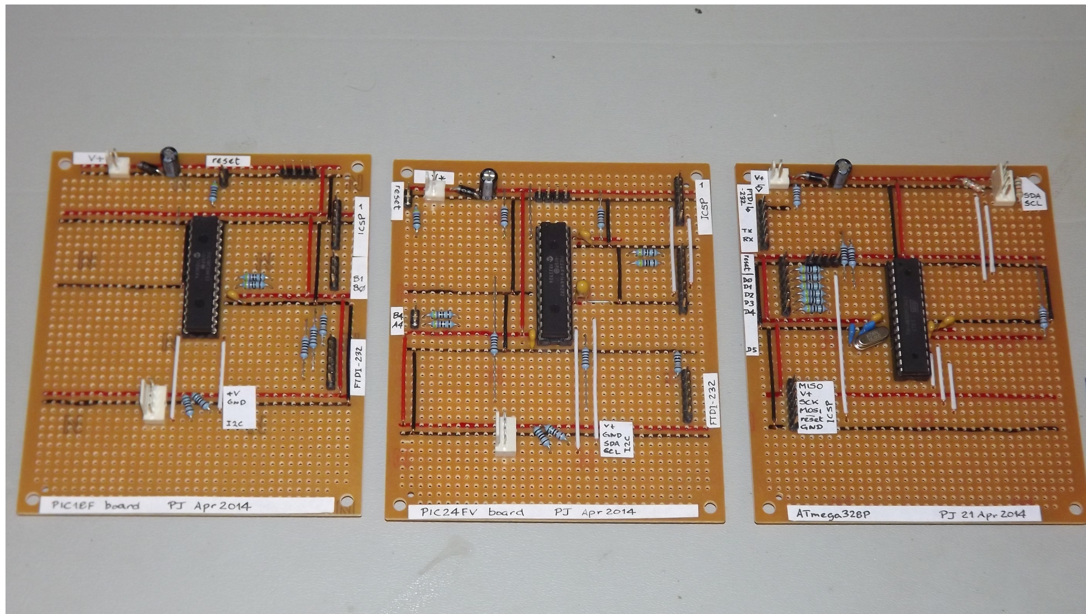
2.2 AVR and PIC24 boards

The Eleven from Freetronics, shown in the left half of the following photograph, is an Arduino-compatible board carrying an ATmega328P microcontroller. This is a convenient piece of hardware with many prototype-friendly boards available to plug into the headers around the periphery of the board. Although these boards come with the Arduino bootloader preprogrammed into the ATmega328 microcontroller, the standard AVR 6-pin programming header on the right-hand end of the board (in the photo) can be used to reprogram the microcontroller with the FlashForth interpreter. Power and serial port access is through the USB connector at the left.

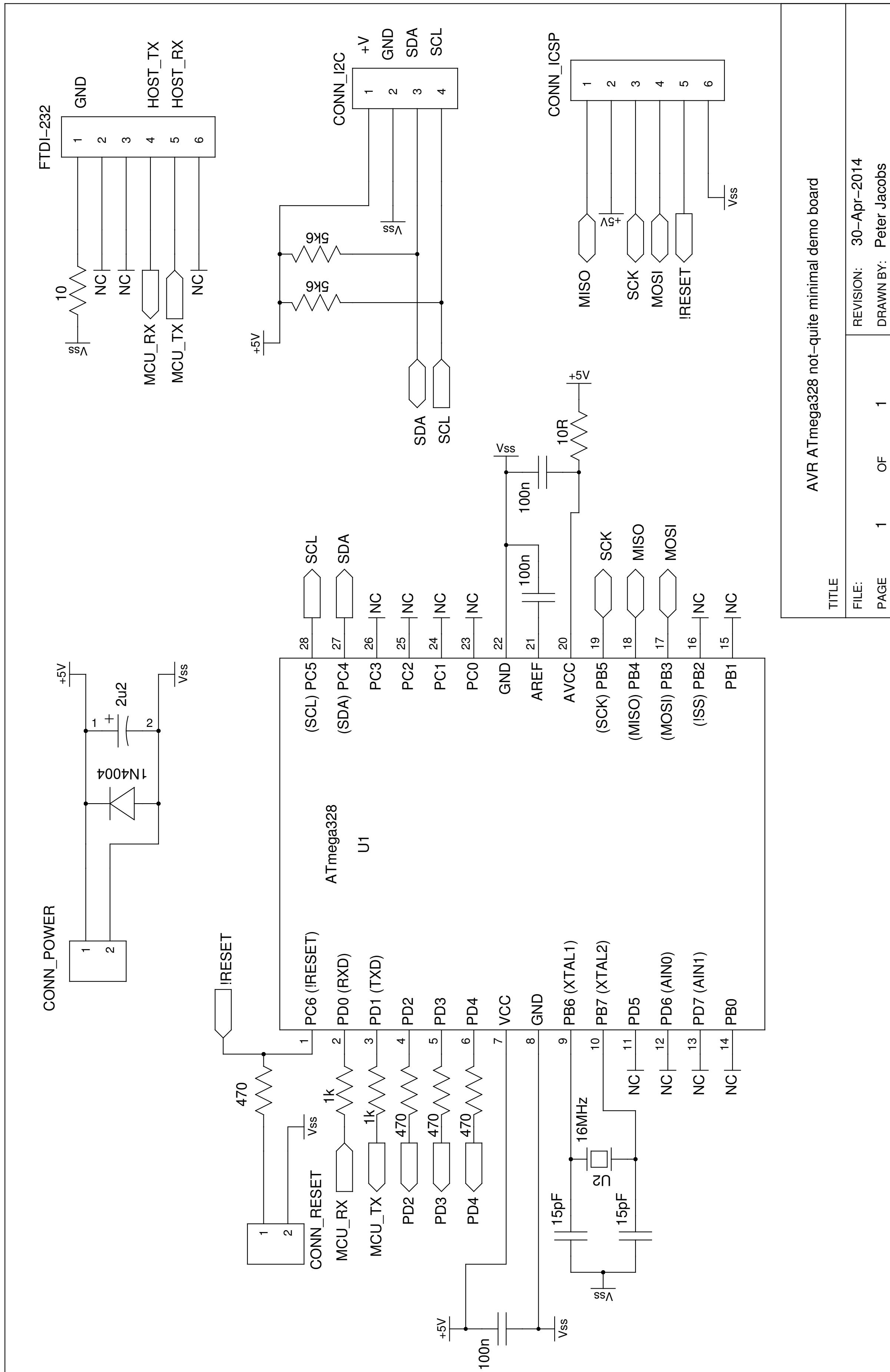
If you want an almost-no-solder option for prototyping with the PIC24FV32KA302, Microchip provide the Microstick 5V for PIC24K-series. As shown in the following photograph, this is convenient in that it includes a programmer on-board and can be plugged into a bread-board. The power supply and flash programming access is provided through the USB connector on the left of the board while the serial port connection is via the 6-pin connector on the right-end of the board.

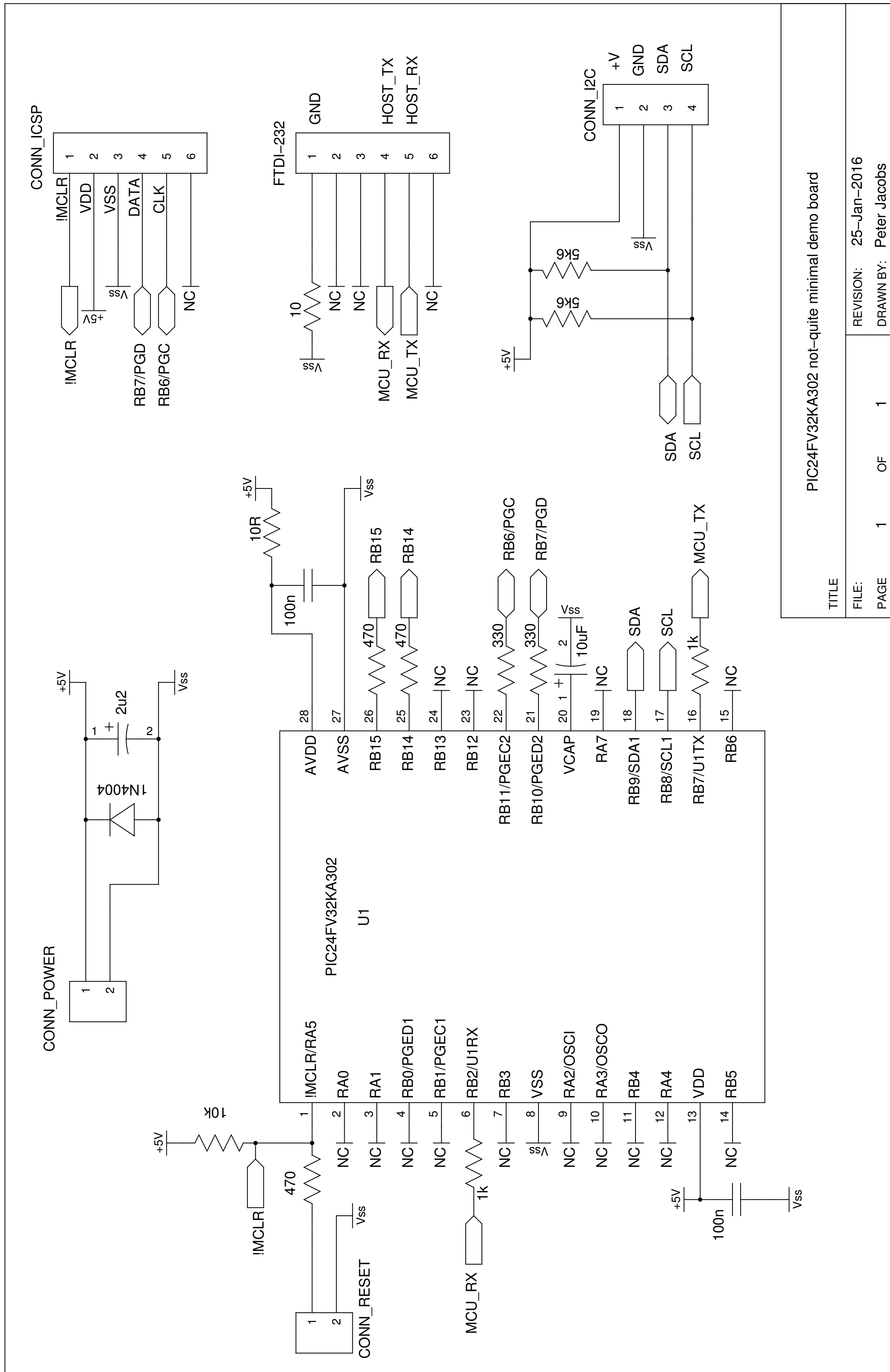


Building a minimal board, by hand, for any of these processors is fairly easy and strip-board versions for each is shown in the following photograph. The left-hand board is for the PIC18F26K22, before all of the extra protection resistors were added. In this state, FlashForth can already be used on this board for nearly all of the exercises in the following sections. Schematic diagrams for the PIC24 and AVR microcontrollers are shown on the following pages.



Each of the boards has headers for (1) power, (2) in-circuit serial programming, (3) I2C communication and (4) TTL-level-232 serial communication. The ATmega328 board on the right has a few more protection resistors installed and has an 16 MHz crystal because serial-port communication was found to be unreliable using the internal oscillator.


AVR ATmega328 not-quite minimal demo board
TITLE
FILE:
PAGE 1 OF 1
REVISION: 30-Apr-2014
DRAWN BY: Peter Jacobs
FTDI-232
GND
HOST_TX
HOST_RX
MCU_RX
MCU_TX
NC
10
Vss
CONN_I2C
+V
GND
SDA
SCL
5k6
+5V
CONN_ICSP
MISO
SCK
MOSI
!RESET
CONN_POWER
1N4004
2u2
ATmega328
U1
(SCL) PC5
(SDA) PC4
PC3
PC2
PC1
PC0
GND
AREF
AVCC
(SCK) PB5
(MISO) PB4
(MOSI) PB3
(!SS) PB2
PB1
PC6 (!RESET)
PD0 (RXD)
PD1 (TXD)
PD2
PD3
PD4
VCC
GND
PB6 (XTAL1)
PB7 (XTAL2)
PD5
PD6 (AIN0)
PD7 (AIN1)
PB0
100n
10R
16MHz
U2
15pF
470
1k
CONN_RESET



CONN_ICSP
!MCLR
VDD
VSS
DATA
CLK
NC
+5V
Vss
RB7/PGD
RB6/PGC
FTDI−232
GND
HOST_TX
HOST_RX
10
MCU_RX
MCU_TX
CONN_I2C
+V
SDA
SCL
5k6
CONN_POWER
2u2
1N4004
+5V
Vss
PIC24FV32KA302
U1
AVDD
AVSS
RB15
RB14
RB13
RB12
RB11/PGEC2
RB10/PGED2
VCAP
RA7
RB9/SDA1
RB8/SCL1
RB7/U1TX
RB6
!MCLR/RA5
RA0
RA1
RB0/PGED1
RB1/PGEC1
RB2/U1RX
RB3
VSS
RA2/OSCI
RA3/OSCO
RB4
RA4
VDD
RB5
10R
100n
470
330
10uF
1k
10k
CONN_RESET
TITLE
PIC24FV32KA302 not−quite minimal demo board
FILE:
REVISION: 25−Jan−2016
DRAWN BY: Peter Jacobs
PAGE 1 OF 1

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 [5] is available online[1], as is Pelc's more recent book [6][2]. A more detailed reference is published by Forth Inc [7]. 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. For an introductory document that is specific to FlashForth, see the companion report [8].

FlashForth [1] for the PIC18, PIC24 and ATmega families of microcontrollers is a full interpreter and compiler that runs entirely on the microcontroller. It is a 16-bit Forth with a byte-addressable memory space. Even though there are distinct memory types (RAM, EEPROM and Flash) and separate busses for data and program memory in these Harvard-architecture microcontrollers, FlashForth unifies them into a single 64kB memory.

Above working in assembler, FlashForth 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 is written in assembler, with one program source for each of the microcontroller families and a number of Forth text files to augment the core interpreter. The source code can be downloaded from SourceForge at the URL
`http://sourceforge.net/projects/flashforth/`
There, you will see that you can get a packaged release or you can clone the git repository.

To build from this source, you will need to start up your integrated development environment (be it MPLAB, MPLAB-X or AVR Studio), open the program source and config files in this IDE and edit the config file(s) match your selection of oscillator. There are other options to customize but the choice of oscillator is the main one. The machine code can then be assembled and programmed into your microcontroller with a suitable device programmer (PICkit3, ICD3, STK500, AVRISP MkII, ...). Once programmed with

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

FlashForth, and mounted in a board that provides power and serial communications as described in the previous section, you will be ready to interact with FlashForth via a serial terminal or shell.

3.2 Building for the PIC18F26K22 or PIC18F46K22

For our minimal system with either the PIC18F26K22 or PIC18F46K22 microcontroller, we elect to use the internal (16 MHz) oscillator multiplied by 4 by the PLL. Within the `MPLAB-X` development environment, we started a new standalone project to build our FlashForth program that will use the microcontroller's UART serial port as the OPERATOR communications channel. Following the prompt screens, we selected a specific processor (PIC18F26K22), our hardware tool (ICD3), and the compiler toolchain (mpasm).

To build the actual machine code that will be programmed into the flash memory of the microcontroller, it is sufficient to assemble the principal source file `ff-pic18.asm` along with the configuration (or header) files `pic18f-main.cfg`, `pic18fxxxx.cfg`, `p18f2x4xk22.cfg`, and use the linker script `FF_0000.lkr`. The source file and config files can be found in the directory `pic18/src/`, while the linker file is in `pic18/lkr/`. There may be other configuration files already added to the project but you can ignore them.

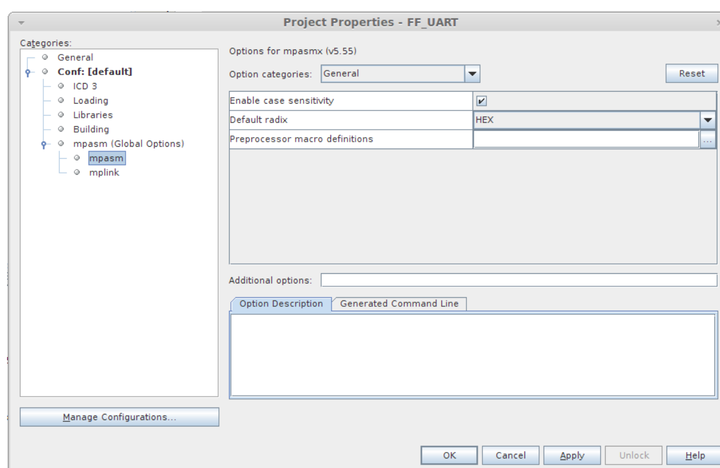
We edited the processor-specific config file, `p18f2x4xk22.cfg`, writing "`PLLCFG = ON`" to have the PLL enabled (giving $F_{OSC} = 64\,\text{MHz}$), enable the watchdog timer with a 1:256 postscale (`WDTPS = 256`) to get approximately a 1 second time-out period, and enable the external reset capability (`MCLRE = EXTMCLR`). Being able to reset the microcontroller by bringing the `MCLR` pin low is something that we find convenient when tinkering with new hardware. We set the final line as

```
#define PLL ENABLE
```

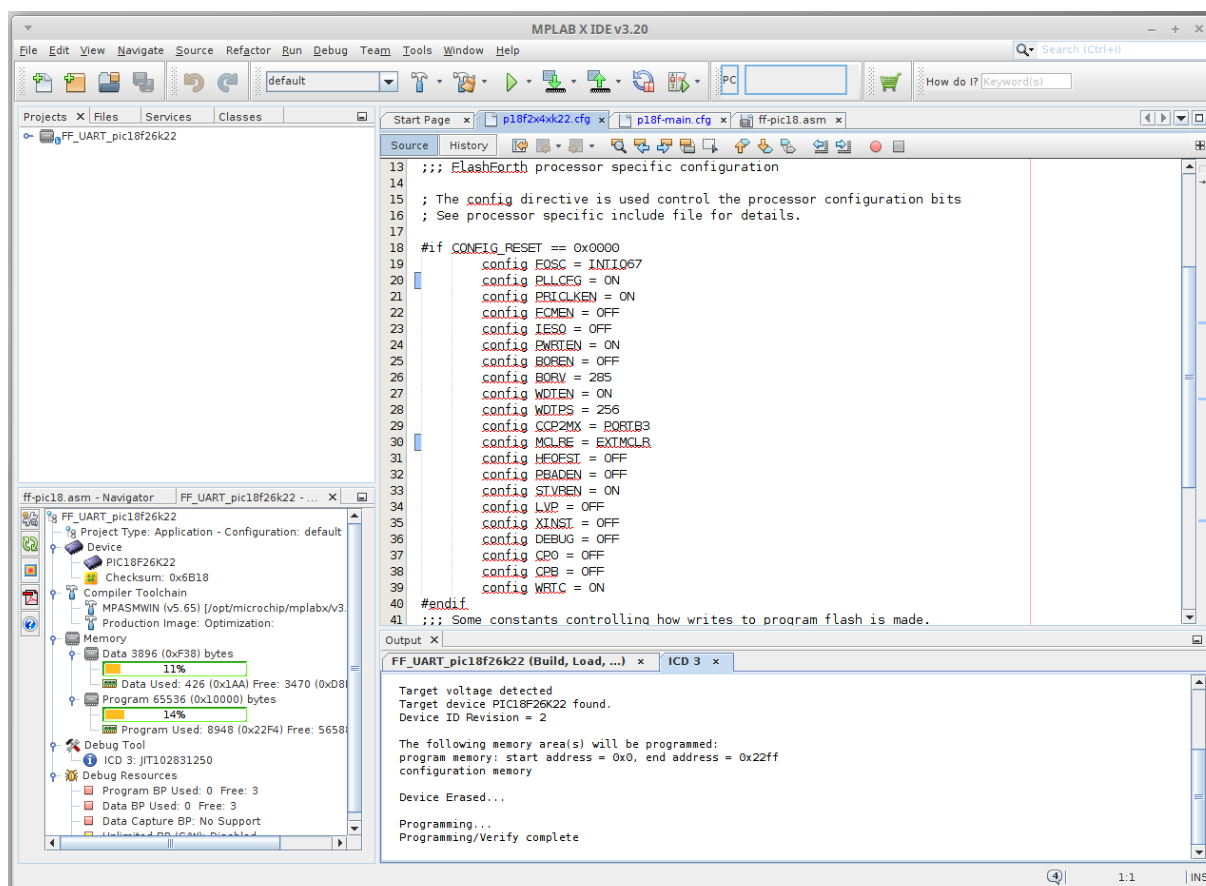
We needed to edit the `pic18f-main.cfg` file only to set the system clock frequency as `constant clock=d'64000000'`. With this clock frequency, the microcontroller requires approximately 7 mA current while the interpreter is running and waiting for input.

There are many other options for customizing the FlashForth program in this file, however, the default parameters are fine for the first build of our minimal system. To see your options for all of the configuration bits for your specific microcontroller, it is convenient to open the MPLAB-X view from the main menu: `Window` → `PIC Memory Views` → `Configuration Bits`.

With the specific microcontroller selected for the project, the config file `pic18fxxxx.cfg` will automatically select the appropriate MPLAB include file for the microcontroller, be it `p18f26k22.inc` for the 28-pin chip on the home-made board or `p18f46k22.inc` for the 40-pin chip on the PICDEM 2 PLUS board. If the build process complains of not being able to find the MCU-specific include file, you may need to adjust the case-sensitivity of the assembler. This check box can be found in the Project Properties dialog, under "General Options" for the `mpasmx` assembler, as shown in the following screen shot.



The following image shows the result of building in Microchip's MPLAB X IDE. The lower left frame in the MPLAB-X window shows the MCU resources used. With 426 bytes of SRAM used (another 3470 free) and 8948 bytes of program memory used (56588 free), For the PIC18F26K22 MCU, FlashForth occupies only about one-seventh of the microcontroller's program memory. Most of the memory is available for the your application. For more details on the SRAM memory map, see "The Hitchhiker's Guide to FlashForth on PIC18 Microcontrollers". There, Mikael Nordman has provided a memory map that shows how the SRAM memory is allocated within the FlashForth system.



The final step is to program the FlashForth machine code into the flash memory of the microcontroller, using whatever device programmer you happen to have plugged into your

development system. The Dashboard view in the screen shot above shows that we have seleted to use of the MPLAB ICD3.

3.3 Building for the PIC24FV32KA302

Building for the 16-bit PIC24 family is similar process. This time look for the source code files in the `pic24/` subdirectory. There are fewer config files but you may need to customize the closest one for your particular processor. Here is the required text in the `p24fk_config.inc` file for our PIC24FV32KA302-I/SP microcontroller using its internal 8 MHz oscillator with 4× PLL and installed on the home-made minimal board:

```
;;; Device memory sizes. Set according to your device.
;;; You can increase the addressable flash range be decreasing the addressable ram.
;;; Below is the setting for max amount of ram for PIC24FV32KA302
.equ FLASH_SIZE,     0x5800  ; Flash size in bytes without the high byte
                             ; See program memory size in the device datasheet.
.equ RAM_SIZE,       0x0800  ; Ram size in bytes
.equ EEPROM_SIZE,    0x0200  ; Eeprom size

; For some reason the normal config macros did not work
           .pushsection __FOSCSEL.sec, code
           .global __FOSCSEL
__FOSCSEL: .pword FNOSC_FRCPLL
           .popsection
; Start additions for FF Tutorial board with PIC24FV32KA30x
           .pushsection __FOSC.sec, code
           .global __FOSC
__FOSC:    .pword OSCIOFNC_OFF
           .popsection
           .pushsection __FICD.sec, code
           .global __FICD
__FICD:    .pword ICS_PGx2
           .popsection
; End additions

.equ FREQ_OSC, (8000000*4)  ;Clock (Crystal)frequency (Hz)
```

Once programmed, FlashForth uses 542 of the microcontroller's 2048 bytes of SRAM and 4544 of the MCU's 11264 words of Flash memory. This leaves most of the memory for your Forth application program. Although this appears to be a lot less than that available in the PIC18F26K22 MCU, this 16-bit MCU has lots of interesting hardware. With instruction cycle frequency of 16 MHz and the interpreter waiting for input, the current consumption is 7.5 mA, approximately the same as for the 8-bit PIC18F26K22.

3.4 Building for the ATmega328P

Assembling the FlashForth program within the AVR Studio IDE is fairly simple but Mike Nordman has made life even simpler for users of Arduino-like hardware by providing a prebuilt `.hex` file that can be programmed into the ATmega328P. Here is the command for doing so with avrdude on a Linux PC.

```
$ sudo avrdude -p m328p -B 8.0 -c avrisp2 -P usb -e \
  -U efuse:w:0x07:m \
  -U hfuse:w:0xda:m \
  -U lfuse:w:0xff:m \
  -U flash:w:ff_uno.hex:i
```

The fuses are set to use the 16 MHz crystal on the Arduino-like board.

4 Interacting with FlashForth

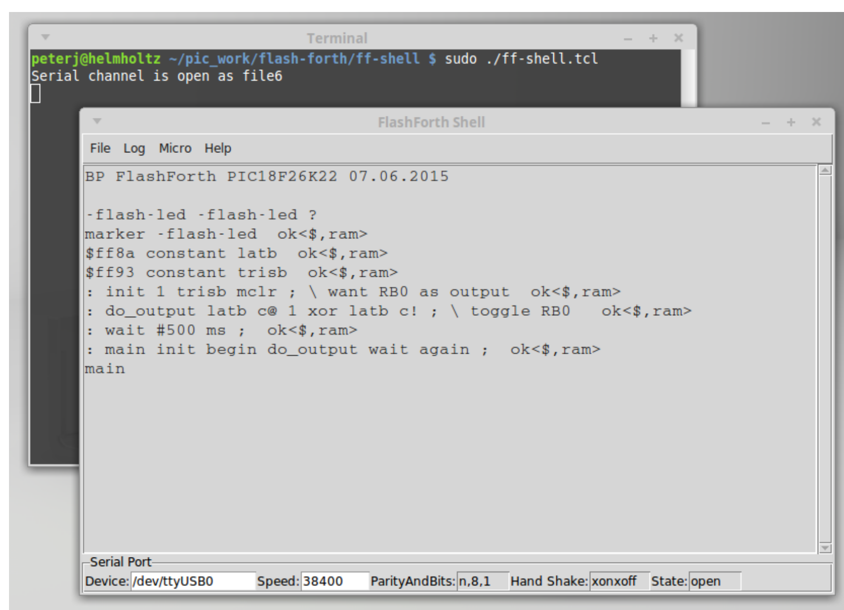
Principally, interaction with the programmed MCU is via the serial port. For the PIC microcontrollers, settings are 38400 baud 8-bit, no parity, 1 stop bit, with software (Xon/Xoff) flow control. For the ATmega328P (as programmed above), the baud rate is 9600.

The FlashForth distribution includes a couple of shell programs that are programmed with some knowledge of the FlashForth interpreter. The `ff-shell.py` program is written in Python and allows interaction with the microcontroller via a standard command shell. It depends on a Python interpreter and the pyserial extension being installed on your PC. The `ff-shell.tcl` is a GUI program that displays the interaction text in a dedicated window on your PC. It requires the Tcl/Tk interpreter which is usually part of a Linux environment but it may be installed on MS-Windows or MacOSX as well.

The following images shows the `ff-shell.tcl` window just afer sending the content of the `flash-led.txt` file to the PIC18F26K22. The device name of `/dev/ttyUSB0` on the status line refers to the USB-to-serial interface that was plugged one of the PC's USB ports. It is convenient to start the program with the command

```
$ sudo ./ff-shell.tcl
```

If necessary, you can adjust the communication settings by typing new values into the entry boxes and pressing `Enter` to repoen the connection.



As you type characters into the main text widget, `ff-shell.tcl` intercepts them and sends them, one at a time, via the serial port to the microcontroller. As the microcontroller sends characters back, the program filters them and displays them in the text widget. There is also a send-file capability that will send the text from the file as fast as it can,

without overwhelming the microcontroller. The Python program `ff-shell.py` has a special command `#send` to start the equivalent process.

If you have sent the microcontroller off to do a repetitive task, such as flashing the LED indefinitely, you can regain the interpreter's attention by sending a `Control-O` character. The interpreter aborts the execution of the current word and does a software restart. After initialization, the interpreter announces that it is ready to begin. Subsequently pressing `Enter` will get the `ok` response, as shown below. The warm restart action is also available from the menu as `Micro`→`Warm Restart`.

```
peterj@helmholtz ~/pic_work/flash-forth/ff-shell $ sudo ./ff-shell.tcl
Serial channel is open as file6
```

```
BP FlashForth PIC18F26K22 07.06.2015

-flash-led -flash-led ?
marker -flash-led  ok<$,ram>
$ff8a constant latb  ok<$,ram>
$ff93 constant trisb  ok<$,ram>
: init 1 trisb mclr ; \ want RB0 as output  ok<$,ram>
: do_output latb c@ 1 xor latb c! ; \ toggle RB0   ok<$,ram>
: wait #500 ms ;  ok<$,ram>
: main init begin do_output wait again ;  ok<$,ram>
main  S FlashForth PIC18F26K22 07.06.2015

  ok<$,ram>
```

Serial Port
Device: /dev/ttyUSB0 Speed: 38400 ParityAndBits: n,8,1 Hand Shake: xonxoff State: open

We find `ff-shell.tcl` a very convenient interaction environment, however, if you want to use a standard terminal program on Linux, Appendix A provides a few notes for doing so.

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 on the PIC18FX6K22

The microcontroller version of the "Hello, World" program is typically a program that flashes a single LED. It will work on either of PIC18F microcontrollers mentioned previously and makes use of a digital input-output pin via the registers that control the IO port. The datasheet [2] has a very readable introduction to the IO ports. Please read it.

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

Notes on this program:

- If the word `-flash-led` has been previously defined with the word marker, line 1 resets the dictionary state and continues interpreting the file, else the interpreter signals that it can't find the word and continues interpreting the file anyway.
- Line 2 records the state of the dictionary and defines the word `-flash-led` so that we can reset the dictionary to its state before the code was compiled, simply by executing the word `-flash-led`.
- Lines 3 and 4 define convenient names for the addresses of the file registers that control IO-port B. Note the literal hexadecimal notation with the `$` character. In the PIC18F family, the special function registers for interacting with the MCU hardware appear near the top of the 64k FlashForth memory space.
- Line 5 is a colon definition for the word `init` that sets up the peripheral hardware. Here, we set pin RB0 as output. The actual command that does the setting is `mclr`, which takes a bit-mask (00000001) and a register address (`$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 6 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 7 defines a word to pause for 500 milliseconds. Note the # character for a literal decimal integer.
- Line 8 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 9 invokes the main word and runs the application. Pressing the Reset button will trigger a hardware restart, kill the application and put the MCU back into a state of listening to the serial port. Invoking a warm restart by typing Control-O or selecting the Warm Restart menu action in ff-shell.tcl may be a more convenient way to stop the application. 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
```

Before going on to more examples, it is good to know about the word empty. This word will reset the dictionary and all of the allotted-memory pointers. Because FlashForth does not allow you to redefine words that are already in the dictionary, later examples that use the same names for their word definitions, may not compile without complaint if you don't clean up after each exercise.

5.2 Flash a LED on the PIC24FV32KA302

```
1 -flash-led
2 marker -flash-led
3 $02c8 constant trisb
4 $02cc constant latb
5 1 #15 lshift constant bit15
6 : init bit15 trisb mclr ; \ set pin as output
7 : do_output latb @ bit15 xor latb ! ; \ toggle the bit
8 : main init begin do_output #500 ms again ;
9 main
```

Notes on this program:

- This program for the 16-bit microcontroller is essentially the same as that for the 8-bit MCU, with different addresses for the port-control registers, of course. In the PIC24/dsPIC30/dsPIC33 version of FlashForth, the special function registers appear in the lowest 2k bytes of memory.

- On line 5, we compute the bit pattern for selecting the MCU pin rather than writing it explicitly. We start with a 1 in the least-significant bit of the 16-bit word and then shift it left 15 places, to produce the binary value `%1000000000000000`
- On line 7, we use 16-bit fetch `@` and store `!` operations because the hardware special function registers on this microcontroller are 16 bits wide.

5.3 Flash a LED on the ATmega328P

```
1  -flash-led-avr
2  marker -flash-led-avr
3  \ PB5 is Arduino digital pin 13.
4  \ There is a LED attached to this pin on the Freetronics Eleven.
5
6  $0024 constant ddrb
7  $0025 constant portb
8  1 #5 lshift constant bit5
9
10 : init bit5 ddrb mset ; \ set pin as output
11 : do_output portb c@ bit5 xor portb c! ; \ toggle the bit
12 : main init begin do_output #500 ms again ;
13
14 main
```

Notes on this program:

- Again, except for the specific registers and bits, this program is the same as for the other MCUs. As for other high-level languages, we no longer have to think about the specific machine architecture (usually).
- Because we are using load and store instructions, the special function registers start at address `$20`.

5.4 Set the cycle duration with a variable (PIC18FX6K22)

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 -flash-led-var
2 marker -flash-led-var
3 \ Flash a LED attached to pin RB0.
4
5 $ff8a constant latb
6 $ff93 constant 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` menu item in `ff-shell.tcl`).
- 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.

5.5 Hello, World: Morse code

Staying with the minimal hardware of just a single LED attached to pin RB0 on the PIC18F26K22 or PIC18F46K22, 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.

```
1 -hello-world
2 marker -hello-world
3 \ Flash a LED attached to pin RB0, sending a message in Morse-code.
4
5 $ff8a constant latb
6 $ff93 constant trisb
7 variable ms_count \ determines the timing.
8
9 : init ( -- )
10   1 trisb mclr  \ want RB0 as output
11   1 latb mclr   \ initial state is off
12 ;
13
14 : led_on 1 latb mset ;
15 : led_off 1 latb mclr ;
16 : gap ms_count @ ms ; \ pause period
17 : gap2 gap gap ;
18 : dit led_on gap led_off gap2 ;
19 : dah led_on gap2 led_off gap2 ;
20
21 \ Have looked up the ARRL CW list for the following letters.
22 : H dit dit dit dit ;
23 : e dit ;
24 : l dit dit ;
25 : o dah dah dah ;
26 : W dit dah dah ;
27 : r dit dah dit ;
28 : d dah dit dit ;
29
30 : greet ( -- )
31   H e l l o  gap  W o r l d  gap2
32 ;
33
34 : main ( n -- )
35   ms_count !  \ store for later use in gap
36   init
37   begin
38     greet
39   again
40 ;
41
42 #100 main  \ exercise the application
```

6 Read and report an analog voltage

6.1 PIC18FX6K22

Use of the analog-to-digital converter (ADC) is a matter of, first, reading Section 17 of the PIC18F2X/4XK22 datasheet [2], 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 -read-adc
2 marker -read-adc
3 \ Read and report the analog value on RA0/AN0.
4
5 \ Registers of interest on the PIC18F26K22
6 $ffc4 constant adresh
7 $ffc3 constant adresl
8 $ffc2 constant adcon0
9 $ffc1 constant adcon1
10 $ffc0 constant adcon2
11 $ff92 constant trisa
12 $ff38 constant ansela
13
14 : init ( -- )
15   1 trisa mset \ want RA0 as input
16   1 ansela mset
17   %00000000 adcon1 c! \ ADC references Vdd, Vss
18   %10101111 adcon2 c! \ right-justified, 12-TAD acq-time, FRC
19   %00000001 adcon0 c! \ Power on ADC, looking at AN0
20 ;
21
22 : adc@ ( -- u )
23   %10 adcon0 mset \ Start conversion
24   begin %10 adcon0 mtst 0= until \ Wait until DONE
25   adresl @
26 ;
27
28 : wait ( -- )
29   #500 ms
30 ;
31
32 : main ( -- )
33   init
34   begin
35     adc@ u.
36     wait
37   key? until
38 ;
39
40 \ Exercise the application, writing digitized values periodically
41 \ until any key is pressed.
42 decimal
43 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 17 to 19 uses binary literals (with the % character) to show the configuration bits explicitly.

- Line 24 conditionally repeats testing of the DONE bit for the ADC.
- Line 25 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 35 invokes the `adc@` word and prints the numeric result.
- Line 37 checks if a character has come in from the serial terminal. If so, the loop is terminated and the main function returns control to the FlashForth interpreter.

6.2 PIC24FV32KA30X

The analog-to-digital converter on the PIC24-series microcontrollers is a little more complex than that on the PIC18 series. There are more features to select and so there are more registers and bits to set, however, the essential set-up tasks are similar. The following script sets up some word definitions that were developed with a view to using them in a larger program. The particular words are more verbose but also carry more information.

```
1 -read-adc
2 marker -read-adc
3 \ Read and report the analog values on AN0 through AN3.
4
5 \ Registers of interest on the PIC24FV32KA30x
6 $0084 constant ifs0
7
8 $02c0 constant trisa
9 $02c2 constant porta
10 $02c4 constant lata
11 $02c6 constant odca
12
13 $02c8 constant trisb
14 $02ca constant portb
15 $02cc constant latb
16 $02ce constant odcb
17
18 $0300 constant adc1buf0
19 $0340 constant ad1con1
20 $0342 constant ad1con2
21 $0344 constant ad1con3
22 $0348 constant ad1chs
23
24 $04e0 constant ansa
25 $04e2 constant ansb
26
27 $0770 constant pmd1
28
29 \ bit masks
30 $0001 constant mADC1MD \ pmd1
31 $0001 constant mDONE \ ad1con1
32 $0002 constant mSAMP
33 $8000 constant mADON
34 $2000 constant mAD1IF
35
36
37 : adc.init ( -- )
38   $0003 trisa mset \ want RA0, RA1 as input
39   $0003 ansa mset
40   $0003 trisb mset
41   $0003 ansb mset
```

```
42   mADC1MD pmd1 mclr \ ensure module enabled
43   $0470 ad1con1 ! \ 12-bit, auto-convert
44   $0000 ad1con2 ! \ ADC references Vdd, Vss
45   $9f00 ad1con3 ! \ ADRC, 31-TAD acq-time
46   $0000 ad1chs ! \ neg input is Vss, pos input AN0
47   mADON ad1con1 mset \ Power on ADC
48   mAD1IF ifs0 mclr
49 ;
50
51 : adc.close ( -- )
52   mADON ad1con1 mclr
53   mAD1IF ifs0 mclr
54 ;
55
56 : adc.select ( u -- ) \ select positive input
57   $0003 and ad1chs ! \ limit selection to AN0 through AN3
58 ;
59
60 : adc@ ( -- u )
61   mDONE ad1con1 mclr
62   mSAMP ad1con1 mset \ Start sampling
63   begin mDONE ad1con1 mtst until \ Wait until done.
64   adc1buf0 @
65 ;
66
67 : adc@.filter ( -- u )
68   0 \ start of sum
69   8 for adc@ + next
70   8 /
71 ;
72
73 : wait ( -- )
74   #500 ms
75 ;
76
77 : adc.test ( -- )
78   adc.init
79   begin
80     0 adc.select adc@.filter u.
81     1 adc.select adc@.filter u.
82     cr
83     wait
84   key? until
85   adc.close
86 ;
87
88 \ Exercise the application, writing digitized values periodically
89 \ until any key is pressed.
90 \ decimal
91 \ adc.test
```

Notes on this program:

- This script was part of a larger application for the monitoring of 2 pressure transducers, hence the setting up of just RA0 and RA1 at the start of `adc.init` at lines 38–41.
- To save power the peripheral modules of a PIC24 are, by default, disabled. You need to clear a module's disable bit (line 42) to do anything with it, even setting configuration registers. The (separate) power-on bit still needs to be set to start up the converter.

7 Counting button presses

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

```
1 \ Use a push-button on RB0 to get user input.
2 \ This button is labelled S3 on the PICDEM2+ board.
3 -pb-demo
4 marker -pb-demo
5
6 $ff81 constant portb
7 $ff8a constant latb
8 $ff93 constant trisb
9
10 variable count
11
12 : init ( -- )
13   %01 trisb mset \ RB0 as input
14   %10 trisb mclr \ RB1 as output
15   %10 latb mclr
16 ;
17 : RB1toggle ( -- )
18   latb c@ %10 xor latb c!
19 ;
20 : RB0@ ( -- c )
21   portb c@ %01 and
22 ;
23 : button? ( -- f )
24   \ Check for button press, with software debounce.
25   \ With the pull-up in place, a button press will give 0.
26   RB0@ if
27     0
28   else
29     #10 ms
30     RB0@ if 0 else -1 then
31   then
32 ;
33
34 : main ( -- )
35   0 count !
36   init
37   begin
38     button? if
39       RB1toggle
40       count @ 1+ count !
41       count @ .
42       #200 ms \ allow time to release button
43     then
44     cwd
45   key? until
46 ;
47
48 main  \ exercise the application
```

Notes on this program:

- The `main` word clears the `count` variable, calls `init` to set up the hardware and then loops, polling `RB0` and incrementing value of the `count` variable only when the button gets pressed.
- If the pause after acknowledging the button press (line 42) is too long, we may lose later button press events. This depends on how frantically we press S3.

- Line 44 resets the watch-dog timer on each pass of the main loop. If we don't press the RB0 button for a long time, the main loop would not otherwise pause and clear the watch-dog timer. The watch-dog timer is cleared inside the `ms` word, however, if the timer expires before being cleared, the microcontroller would be reset and the FlashForth interpreter would restart.

8 Counting button presses via interrupts

Instead of polling the RB0 pin attached to the push button, as in the previous example, let's set up the hardware interrupt mechanism to invoke the increment action for us.

```
1  \ Use a push-button on RB0 to get user input, via an interrupt.
2  \ This button is labelled S3 on the PICDEM2+ board.
3  \ Don't have J6 connected because the LED on RB0 loads the pull-up.
4
5  -pb-interrupt
6  marker -pb-interrupt
7
8  $ff93 constant trisb
9  $fff2 constant intcon
10 $fff1 constant intcon2
11
12 variable count
13 variable last-count
14
15 : int0-irq
16   [i
17     %10 intcon mtst \ INT0IF
18     if
19       count @ 1+ count !
20       %10 intcon mclr
21     then
22   i]
23 ;i
24
25 : init ( -- )
26   %01 trisb mset \ RB0 as input, a button press will give 0.
27   %01000000 intcon2 mclr \ interrupt on falling edge
28   ['] int0-irq 0 int! \ install service word
29   %10 intcon mclr \ INT0IF cleared
30   %10000 intcon mset \ INT0 interrupt enable
31 ;
32
33 : main ( -- )
34   0 count !
35   init
36   begin
37     count @ last-count @ - \ change?
38     if
39       count @ dup last-count ! .
40     then
41     cwd
42   key? until
43 ;
44
45 main  \ exercise the application
```

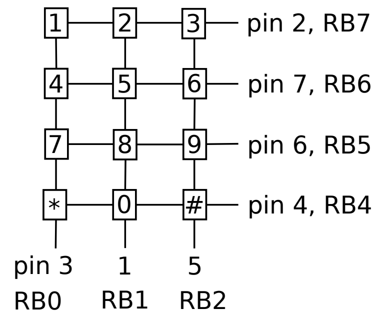
Notes on this program:

- Again, we use the variable named `count` as the variable to be incremented on pressing the button that pulls RB0 low. The actual increment is done on line 19, inside the interrupt service word `int0-irq`. The second variable, `last-count`, is used on line 36 in the `main` word, to detect when the `count` variable changes.
- The `init` word sets up the bits to enable the `INT0` external interrupt to fire on a falling edge at RB0.

- On line 28 in the `init` word, the execution token for our interrupt service word is stored as the high-priority interrupt vector. Because FlashForth supports only high-priority interrupts, the `0` is a dummy value but is still expected by the `int!` word.
- Inside the interrupt-service word, we need to test the `INT0IF` interrupt flag to see if it is our interrupt to handle and, if it is, do the appropriate work (of incrementing the `count` variable) and clearing the interrupt flag. If you enable several interrupt sources, you need to provide a test and action for each.
- The `main` word clears the `count` variable, calls `init` to set up the interrupt mechanism and then loops, emitting the value of the `count` variable only when it changes.

9 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.



To minimize hardware, we have used the weak pull-ups on PORTB. Pressing a key while its 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.

```
1 -keypad
2 marker -keypad
3 \ Display key presses from a 4x3 (telephone-like) keypad
4 \ on PIC18F26K22-I/SP
5
6 $ff81 constant portb
7 $ff8a constant latb
8 $ff93 constant trisb
9 $ff39 constant anselb
10 $ff61 constant wpub
11 $fff1 constant intcon2
12
13 : init ( -- )
14   0 latb c!
15   %00000000 anselb c!    \ set as all digital I/O pins
16   %11110000 trisb c!     \ RB7-4 as input, RB3-0 as output
17   %11110000 wpub c!      \ pull-ups on RB7-4
18   %10000000 intcon2 mclr \ turn on pull-ups
19 ;
20
21 flash
22 create key_chars
23   char 1 c, char 2 c, char 3 c,
24   char 4 c, char 5 c, char 6 c,
25   char 7 c, char 8 c, char 9 c,
26   char * c, char 0 c, char # c,
27 create key_scan_bytes
28   $7e c, $7d c, $7b c,
29   $be c, $bd c, $bb c,
30   $de c, $dd c, $db c,
31   $ee c, $ed c, $eb c,
32 ram
33
34 : scan_keys ( -- c )
35   \ Return ASCII code of key that is pressed
36   #12 for
37     key_scan_bytes r@ + c@
38     dup
39     latb c!
40     portb c@
41     = if
42       \ key must be pressed to get a match
43       key_chars r@ + c@
44       rdrop
```

```
45       exit
46     then
47   next
48   0  \ no key was pressed
49 ;
50
51 : keypad@ ( -- c )
52   \ Read keypad with simple debounce.
53   \ ASCII code is left on stack.
54   \ Zero is returned for no key pressed or inconsistent scans.
55   scan_keys dup
56   #20 ms
57   scan_keys
58   = if exit else drop then
59   0 \ inconsistent scan results
60 ;
61
62 : main ( -- )
63   init
64   begin
65     keypad@
66     dup
67     0= if
68       drop  \ no key pressed
69     else
70       emit
71       #300 ms \ don't repeat key too quickly
72     then
73   key? until
74 ;
```

Notes on this program:

- In lines 21–31, 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 36 and 47 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.

10 Base words for an I^{2}C master

Here are some words for using I^{2}C (or Two-wire) peripherals for each of the microcontrollers in master mode. These words provide abstract the hardware registers and bits to provide a common vocabulary for the interaction with I^{2}C slave devices.

10.1 PIC18FX6K22

```
1 \ i2c-base-k22.txt
2 \ Low-level words for I2C master on PIC18F26K22
3 \
4 \ Modelled on the original i2c-base.txt for PIC18,
5 \ i2c-twi.frt from amforth and
6 \ the datasheet for Microchip PIC18F26K22.
7 \ Peter J.  2014-11-08
8
9 -i2c-base-k22
10 marker -i2c-base-k22
11 hex ram
12
13 \ Registers related to I2C operation of MSSP1
14 $ff3a constant anselc
15 $ff82 constant portc
16 $ff8b constant latc
17 $ff94 constant trisc
18 $ff9e constant pir1
19 $ffc5 constant ssp1con2
20 $ffc6 constant ssp1con1
21 $ffc7 constant ssp1stat
22 $ffc8 constant ssp1add
23 $ffc9 constant ssp1buf
24 $ffca constant ssp1msk
25 $ffcb constant ssp1con3
26
27 \ Masks for bits
28 %00000001 constant mSEN \ in ssp1con2
29 %00000010 constant mRSEN
30 %00000100 constant mPEN
31 %00001000 constant mRCEN
32 %00010000 constant mACKEN
33 %00100000 constant mACKDT
34 %01000000 constant mACKSTAT
35 %00100000 constant mSSP1EN \ in ssp1con1
36 %00000001 constant mBF \ in ssp1stat
37 %00001000 constant mSSP1IF \ in pir1
38
39 : i2c.init ( -- )
40   %00001000 ssp1con1 c! \ Master mode
41   [ Fcy #100 / 1- ] literal ssp1add c! \ Set clock frequency to 100 kHz
42   mSSP1IF pir1 mclr \ Clear interrupt bit
43   %00011000 trisc mset \ SCL1 on RC3, SDA1 on RC4
44   %00011000 anselc mclr
45   mSSP1EN ssp1con1 mset \ Enable hardware
46 ;
47
48 : i2c.close ( -- )
49   mSSP1EN ssp1con1 mclr
50   mSSP1IF pir1 mclr
51 ;
52
53 : i2c.wait ( -- ) \ Wait for interrupt flag and clear it
54   begin mSSP1IF pir1 mtst until
55   mSSP1IF pir1 mclr
56 ;
57
58 : i2c.idle? ( -- f )
```

```
59   %00011111 ssp1con2 mtst \ ACKEN RCEN REN RSEN SEN
60   %100 ssp1stat mtst \ R/^W
61   or 0=
62 ;
63
64 : i2c.start ( -- ) \ Send start condition
65   begin i2c.idle? until
66   mSSP1IF pir1 mclr
67   mSEN ssp1con2 mset
68   i2c.wait
69 ;
70
71 : i2c.rsen ( -- ) \ Send repeated start condition
72   mSSP1IF pir1 mclr
73   mRSEN ssp1con2 mset
74   i2c.wait
75 ;
76
77 : i2c.stop ( -- ) \ Send stop condition
78   mSSP1IF pir1 mclr
79   mPEN ssp1con2 mset
80   i2c.wait
81 ;
82
83 : i2c.buf.full? ( -- f )
84   mBF ssp1stat mtst
85 ;
86
87 \ Write one byte to bus, leaves ACK bit.
88 \ A value of 0 indicates ACK was received from slave device.
89 : i2c.c! ( c -- f )
90   begin i2c.buf.full? 0= until
91   ssp1buf c!
92   begin i2c.buf.full? 0= until
93   begin i2c.idle? until
94   ssp1con2 c@ mACKSTAT and
95 ;
96
97 \ Send ack bit.
98 : i2c.ack.seq ( -- )
99   mACKEN ssp1con2 mset
100  begin mACKEN ssp1con2 mtst 0= until
101 ;
102
103 \ Read one byte and ack for another.
104 : i2c.c@.ack ( -- c )
105   mRCEN ssp1con2 mset
106   begin i2c.buf.full? until
107   mACKDT ssp1con2 mclr i2c.ack.seq \ ack
108   ssp1buf c@
109 ;
110
111  \ Read one last byte.
112 : i2c.c@.nack ( -- c )
113   mRCEN ssp1con2 mset
114   begin i2c.buf.full? until
115   mACKDT ssp1con2 mset i2c.ack.seq \ nack
116   ssp1buf c@
117 ;
118
119 \ Address slave for writing, leaves true if slave ready.
120 : i2c.addr.write ( 7-bit-addr -- f )
121   1 lshift 1 invert and \ Build full byte with write-bit as 0
122   i2c.start i2c.c! 0=
123 ;
124
125 \ Address slave for reading, leaves true if slave ready.
126 : i2c.addr.read ( 7-bit-addr -- f )
127   1 lshift 1 or \ Build full byte with read-bit as 1
128   i2c.start i2c.c! 0=
129 ;
```

```
130
131 \ Detect presence of device, leaving true if device present, 0 otherwise.
132 \ We actually fetch a byte if the slave has acknowledged, then discard it.
133 : i2c.ping? ( 7-bit-addr -- f )
134   i2c.addr.read if i2c.c@.nack drop true else false then
135 ;
```

10.2 PIC24FV32KA30X

```
1 \ i2c-base-pic24fv32ka30x.txt
2 \ Low-level words for I2C master on PIC24FV32KA302 and KA301
3 \
4 \ Modelled on i2c-base.txt for PIC18, i2c-twi.frt from amforth
5 \ the Microchip PIC24 Family Reference Manual
6 \ and the datasheet for PIC24FV32KA304 family.
7 \ Peter J.  2015-09-23
8
9 -i2c-base
10 marker -i2c-base
11 hex ram
12
13 \ Registers related to I2C operation of MSSP1
14 $0086 constant ifs1
15 $0200 constant i2c1rcv
16 $0202 constant i2c1trn
17 $0204 constant i2c1brg
18 $0206 constant i2c1con
19 $0208 constant i2c1stat
20 $020a constant i2c1add
21 $020c constant i2c1msk
22 $02c8 constant trisb
23 $02ca constant portb
24 $02cc constant latb
25 $02ce constant odcb
26 $04e2 constant ansb
27 $0770 constant pmd1
28
29 \ Masks for bits
30 $8000 constant mI2CEN \ in i2c1con
31 %000001 constant mSEN
32 %000010 constant mRSEN
33 %000100 constant mPEN
34 %001000 constant mRCEN
35 %010000 constant mACKEN
36 %100000 constant mACKDT
37 $8000 constant mACKSTAT \ in i2c1stat
38 $4000 constant mTRSTAT
39 $0400 constant mBCL
40 $0080 constant mIWCOL
41 $0040 constant mI2COV
42 %0001 constant mTBF
43 %0010 constant mRBF
44 %0010 constant mMI2C1IF \ in ifs1
45
46 $0100 constant mRB8 \ SCL1 on RB8
47 $0200 constant mRB9 \ SDA1 on RB9
48
49 : i2c.init ( -- )
50   $80 pmd1 mclr \ Enable the I2C1 module
51   [ Fcy #100 / Fcy #10000 / - 1- ] literal i2c1brg c! \ Set clock to 100 kHz
52   mMI2C1IF ifs1 mclr \ Clear interrupt bit for master operation
53   %1100000000 trisb mset \ SCL1 on RB8, SDA1 on RB9
54   %1100000000 odcb mset
55   mI2CEN i2c1con mset \ Enable hardware
```

```
56 ;
57
58 : i2c.close ( -- )
59   mI2CEN i2c1con mclr
60   mMI2C1IF ifs1 mclr
61 ;
62
63 : i2c.bus.reset ( -- )
64   \ Manually reset the slave devices.
65   \ For use when a slave just won't let SDA1 go.
66   i2c.close
67   mRB9 trisb mset \ leave SDA1 float
68   mRB9 odcb mset
69   mRB8 trisb mclr \ drive SCL1 with digital output
70   mRB8 odcb mset
71   9 for
72     mRB8 latb mclr 1 ms
73     mRB8 latb mset 1 ms
74   next
75   \ stop condition
76   mRB8 latb mclr
77   mRB9 trisb mclr
78   mRB9 latb mclr 1 ms
79   mRB8 latb mset
80   mRB9 latb mset 1 ms
81   \ release bus
82   mRB8 trisb mset
83   mRB9 trisb mset
84 ;
85
86 : i2c.wait ( -- ) \ Wait for interrupt flag and clear it
87   begin mMI2C1IF ifs1 mtst until
88   mMI2C1IF ifs1 mclr
89 ;
90
91 : i2c.idle? ( -- f )
92   %00011111 i2c1con mtst \ ACKEN RCEN REN RSEN SEN
93   0=
94 ;
95
96 : i2c.start ( -- ) \ Send start condition
97   begin i2c.idle? until
98   mMI2C1IF ifs1 mclr
99   mSEN i2c1con mset
100   i2c.wait
101 ;
102
103 : i2c.rsen ( -- ) \ Send repeated start condition
104   mMI2C1IF ifs1 mclr
105   mRSEN i2c1con mset
106   i2c.wait
107 ;
108
109 : i2c.stop ( -- ) \ Send stop condition
110   mMI2C1IF ifs1 mclr
111   mPEN i2c1con mset
112   i2c.wait
113 ;
114
115 : i2c.tbuf.full? ( -- f )
116   mTBF i2c1stat mtst
117 ;
118
119 : i2c.rbuf.full? ( -- f )
120   mRBF i2c1stat mtst
121 ;
122
123 \ Write one byte to bus, leaves ACK bit.
124 \ A value of 0 indicates ACK was received from slave device.
125 : i2c.c! ( c -- f )
126   begin i2c.tbuf.full? 0= until
```

```
127   mMI2C1IF ifs1 mclr
128   i2c1trn c!
129   \ We wait for the interrupt because just waiting for the buffer
130   \ to be empty is unreliable if we look too soon.
131   i2c.wait
132   begin i2c.idle? until
133   i2c1stat @ mACKSTAT and
134 ;
135
136 \ Send ack bit.
137 : i2c.ack.seq ( -- )
138   mACKEN i2c1con mset
139   begin mACKEN i2c1con mtst 0= until
140 ;
141
142 \ Read one byte and ack for another.
143 : i2c.c@.ack ( -- c )
144   mRCEN i2c1con mset
145   begin i2c.rbuf.full? until
146   mACKDT i2c1con mclr i2c.ack.seq \ ack
147   i2c1rcv c@
148 ;
149
150  \ Read one last byte.
151 : i2c.c@.nack ( -- c )
152   mRCEN i2c1con mset
153   begin i2c.rbuf.full? until
154   mACKDT i2c1con mset i2c.ack.seq \ nack
155   i2c1rcv c@
156 ;
157
158 \ Address slave for writing, leaves true if slave ready.
159 : i2c.addr.write ( 7-bit-addr -- f )
160   1 lshift 1 invert and \ Build full byte with write-bit as 0
161   i2c.start i2c.c! 0=
162 ;
163
164 \ Address slave for reading, leaves true if slave ready.
165 : i2c.addr.read ( 7-bit-addr -- f )
166   1 lshift 1 or \ Build full byte with read-bit as 1
167   i2c.start i2c.c! 0=
168 ;
169
170 \ Detect presence of device,
171 \ leaving true if device present, 0 otherwise.
172 \ We actually fetch a byte if the slave has acknowledged.
173 : i2c.ping? ( 7-bit-addr -- f )
174   i2c.addr.read if i2c.c@.nack drop true else false then
175 ;
```

10.3 ATmega328P

```
 1 \ i2c-base-avr.txt
 2 \ Low-level words for TWI/I2C on Atmega328P.
 3 \
 4 \ Modelled on i2c-twi.frt from amforth,
 5 \ i2c_base.txt for FlashForth on PIC18
 6 \ and the Atmel datasheet, of course.
 7 \ Peter J.  2014-10-27
 8
 9 -i2c-base
10 marker -i2c-base
11 hex ram
12
```

```
13 \ Two-Wire-Interface Registers
14 $b8 constant TWBR
15 $b9 constant TWSR
16 $bb constant TWDR
17 $bc constant TWCR
18
19 \ Bits in the Control Register
20 %10000000 constant mTWINT
21 %01000000 constant mTWEA
22 %00100000 constant mTWSTA
23 %00010000 constant mTWSTO
24 %00001000 constant mTWWC
25 %00000100 constant mTWEN
26 %00000001 constant mTWIE
27
28 : i2c.init ( -- ) \ Set clock frequency to 100kHz
29   %11 TWSR mclr \ prescale value = 1
30   [ Fcy #100 / #16 - 2/ ] literal TWBR c!
31   mTWEN TWCR mset
32 ;
33
34 : i2c.wait ( -- ) \ Wait for operation to complete
35   \ When TWI operations are done, the hardware sets
36   \ the TWINT interrupt flag, which we will poll.
37   begin TWCR c@ mTWINT and until
38 ;
39
40 : i2c.start ( -- ) \ Send start condition
41   [ mTWINT mTWEN or mTWSTA or ] literal TWCR c!
42   i2c.wait
43 ;
44
45 : i2c.rsen ( -- ) \ Send repeated start condition
46   i2c.start \ AVR doesn't distinguish
47 ;
48
49 : i2c.stop ( -- ) \ Send stop condition
50   [ mTWINT mTWEN or mTWSTO or ] literal TWCR c!
51 ;
52
53 \ Write one byte to bus, returning 0 if ACK was received, -1 otherwise.
54 : i2c.c! ( c -- f )
55   i2c.wait \ Must have TWINT high to write data
56   TWDR c!
57   [ mTWINT mTWEN or ] literal TWCR c!
58   i2c.wait
59   \ Test for arrival of an ACK depending on what was sent.
60   TWSR c@ $f8 and $18 xor 0= if 0 exit then \ SLA+W
61   TWSR c@ $f8 and $28 xor 0= if 0 exit then \ data byte
62   TWSR c@ $f8 and $40 xor 0= if 0 exit then \ SLA+R
63   -1 \ Something other than an ACK resulted
64 ;
65
66 \ Read one byte and ack for another.
67 : i2c.c@.ack ( -- c )
68   [ mTWINT mTWEN or mTWEA or ] literal TWCR c!
69   i2c.wait
70   TWDR c@
71 ;
72
73 \ Read one last byte.
74 : i2c.c@.nack ( -- c )
75   [ mTWINT mTWEN or ] literal TWCR c!
76   i2c.wait
77   TWDR c@
78 ;
79
80 \ Address slave for writing, leaving true if slave ready.
81 : i2c.addr.write ( 7-bit-addr -- f )
82   1 lshift 1 invert and \ Build full byte with write-bit as 0
83   i2c.start i2c.c! if false else true then
```

```
84 ;
85
86 \ Address slave for reading, leaving true if slave ready.
87 : i2c.addr.read ( 7-bit-addr -- f )
88   1 lshift 1 or \ Build full byte with read-bit as 1
89   i2c.start i2c.c! if false else true then
90 ;
91
92 \ Detect presence of device, leaving true if slave responded.
93 \ If the slave ACKs the read request, fetch one byte only.
94 : i2c.ping? ( 7-bit-addr -- f )
95   1 lshift 1 or \ Build full byte with read-bit as 1
96   i2c.start i2c.c! 0= if i2c.c@.nack drop true else false then
97 ;
```

10.4 Notes on using the words

- The word `i2c.init` is used to set up the I^{2}C master peripheral for further activities.
- I^{2}C conversations begin by addressing a slave device for either reading or writing. The words `i2c.addr.read` and `i2c.addr.write` are provided for this waking of the slave. They leave a flag on the stack to indicate whether the slave device acknowledged being addressed. If the slave device responded appropriately, you may proceed to read or write bytes.
- There are two words for reading a byte from the bus. `i2c.c@.ack` reads a byte and asserts an acknowledge (ACK) to indicate to the slave device that another byte will be read subsequently. `i2c.c@.nack` reads a byte and asserts a NACK to indicate to the slave that no more bytes are wanted.
- The word to sending a byte to the slave device is `i2c.c!`. This word leaves a flag to indicate the state of the ACK bit following the action of sending the byte. If the slave asserted ACK, the flag will be `0`. You may `drop` this flag if it not of interest to you.
- There are lower-level words `i2c.start`, `i2c.rsen` and `i2c.stop` to assert start, restart and stop conditions respectively. These are used within the higher-level words mentioned above.
- The utility word `i2c.ping?` attempts to address a slave and read a byte. It leaves `true` if the slave responds, else `false`.
- Sometimes when tinkering with a new I^{2}C device, you can get into a state of confusion such that the slave device will end up in some intermediate state waiting for clock signals.[3] In this state, the slave device will no longer respond in a way that the master peripheral understands. Rather than cycle the power to reset the slave device, it may be convenient to force the clocking of the data bits through the bus and get the slave device back into an idle state. The word `i2c.reset.bus` (in i2c-base-pic24fv32ka30x.txt) is provided to automate this forced clocking.

[3] This happens more often than I would like to admit.

10.5 Detecting I^{2}C devices

Building on the base words for a particular microcontroller, the following program works on all of the microcontrollers discussed in this tutorial guide. It is convenient to run this program to to see if the device of interest is responding. There's no point trying to have a conversation with a device that doesn't respond to being addressed.

```
1 \ i2c-detect.txt
2 \ Detect presence of all possible devices on I2C bus.
3 \ Only the 7 bit address schema is supported.
4 \
5 \ Copied from amForth distribution (lib/hardware/)
6 \ and lightly edited to suit FlashForth 5.0 on AVR.
7 \ Builds upon i2c-base-xxxx.txt and doloop.txt.
8 \ Peter J. 2014-10-27
9
10 -i2c-detect
11 marker -i2c-detect
12
13 \ not all bitpatterns are valid 7bit i2c addresses
14 : i2c.7bitaddr? ( a -- f)  $7 $78 within ;
15
16 : i2c.detect   ( -- )
17     base @ hex
18     \ header line
19     cr 5 spaces $10 0 do i 2 u.r loop
20     $80 0 do
21       i $0f and 0= if
22         cr i 2 u.r [char] : emit space
23       then
24       i i2c.7bitaddr? if
25         i i2c.ping? if \ does device respond?
26             i 2 u.r
27           else
28             ." -- "
29         then
30       else
31          ."    "
32       then
33     loop
34     cr base !
35 ;
36
37 \ With a lone Microchip TC74A0 sitting on the bus,
38 \ the output looks like
39 \ i2c.init  ok<$,ram>
40 \ i2c.detect
41 \      00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
42 \ 00 :                      -- -- -- -- -- -- -- -- --
43 \ 10 : -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
44 \ 20 : -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
45 \ 30 : -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
46 \ 40 : -- -- -- -- -- -- -- -- 48 -- -- -- -- -- -- --
47 \ 50 : -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
48 \ 60 : -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
49 \ 70 : -- -- -- -- -- -- -- --
50 \  ok<$,ram>
51 \ i2c.stop  ok<$,ram>
```

11 Using I²C to get temperature measurements

Using the words in `i2c-base-k22.txt` to control the MSSP peripheral in master mode, one may talk to the TC74A5 temperature measurement chip on the PICDEM 2 PLUS and report sensor temperature.

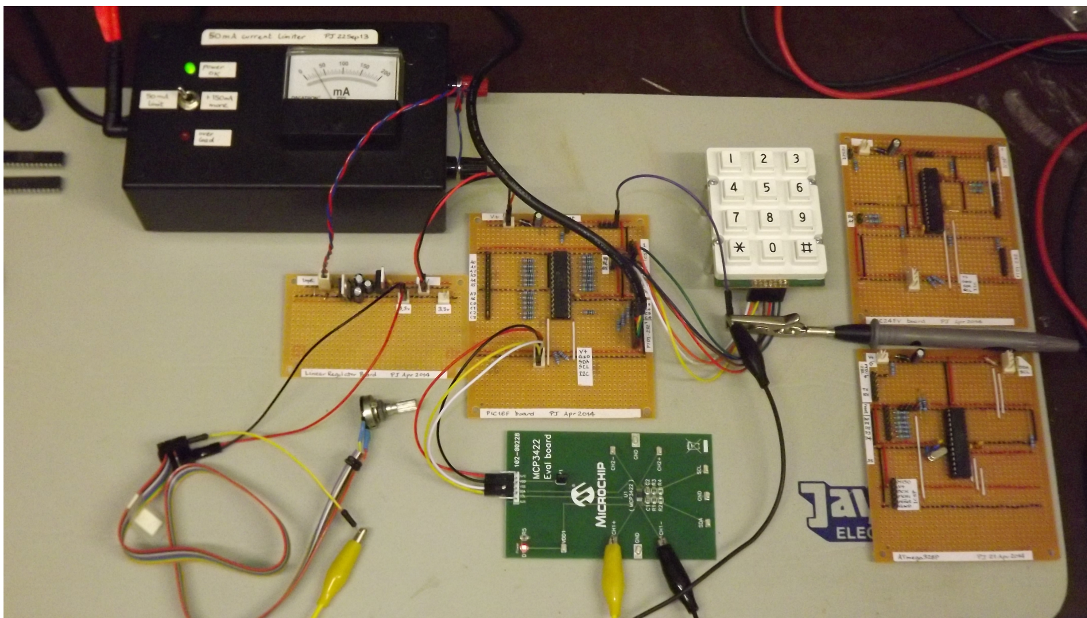
```
\ Read temperature from TC74 on PICDEM2+ board with PIC18F46K22-I/P.
\ Modelled on Mikael Nordman's i2c_tcn75.txt.
\ This program requires i2c-base-k22.txt to be previously loaded.
-read-tc74
marker -read-tc74

%1001101 constant addr-TC74A5  \ 7-bit address for the chip

: tc74-init ( -- )
  \ Selects the temperature register for subsequent reads.
  addr-TC74A5 i2c.addr.write if 0 i2c.c! drop then i2c.stop
;
: sign-extend ( c -- n )
  \ If the TC74 has returned a negative 8-bit value,
  \ we need to sign extend to 16-bits with ones.
  dup $7f > if $ff80 or then
;
: degrees@ ( -- n )
  \ Wake the TC74 and receive its register value.
  addr-TC74A5 i2c.addr.read if i2c.c@.nack sign-extend else 0 then
;
: tc74-main ( -- )
  i2c.init
  tc74-init
  begin
    degrees@ .
    #1000 ms
  key? until
;

\ Now, report temperature in degrees C
\ while we warm up the TC74 chip with our fingers...
decimal tc74-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 `tc74-init` 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.

12 Making high-resolution voltage measurements

The Microchip MCP3422 is a ΣΔ-ADC that can connected via I^{2}C port. This neat little converter can measure voltages with a resolution of 18 bits (at the lowest data rate of 3.75 samples per second) and includes a programmable gain amplifier [9]. Being available in a surface-mount package only, it was convenient to use a prebuilt evaluation board, the green board between the home-built FlashForth demo board and the fixed-voltage supply board. The MCP3422 evaluation board is connected to and powered from the I^{2}C header on the FlashForth demo board. Separately, the fixed-voltage supply board provides the measurement voltage to channel 1 of the MCP3422 via a potentiometer that is set to give 1.024 V, according to my (fairly cheap) multimeter.




```
1 \ mcp3422-2016.txt
2 \ Play with mcp3422 eval board.
3 \ PJ, 21-Oct-2013
4 \     28-Apr-2014 PIC18F26K22 version
5 \     27-Jan-2016 update to use latest i2c words
6 \ Needs i2c-base-k22.txt and math.txt (to get m*/).
7
8 -mcp3422
9 marker -mcp3422
10
11 $68 constant addr-mcp3422 \ 7-bit address
12
13 : mcp3422-init ( -- )
14   \ $9c is config for 18-bit continuous conversions of ch 1
15   addr-mcp3422 i2c.addr.write if $9c i2c.c! drop then i2c.stop
16 ;
17
18 : mcp3422@ ( -- d f ) \ Read the 18-bit result as 3 bytes
19   addr-mcp3422 i2c.addr.read
20   if
21     i2c.c@.ack                       \ only 2 bits in first byte
22     dup $3 > if $fffa or then        \ sign-extend to full cell
23     i2c.c@.ack $8 lshift i2c.c@.ack or \ next two bytes into one cell
24     swap                             \ leave double result
25     i2c.c@.nack $80 and 0=           \ leave true if result is latest
26   else
27     0 0 0  \ device did not ack on address
```

```
28   then
29 ;
30
31 : microvolts ( d1 -- d2 )
32   \ The least-significant bit corresponds to 15.625 microvolts
33   #125 #8 m*/
34 ;
35
36 : (d.3) ( d -- )
37   swap over dabs
38   <# # # # [char] . hold #s rot sign #>
39 ;
40
41 : report ( d f -- ) \ Assuming decimal, print millivolt value
42   cr if ." new " else ." old " then
43   microvolts (d.3) type space ." mV "
44 ;
45
46 : mcp3422-run ( -- )
47   decimal
48   i2c.init mcp3422-init
49   begin
50     mcp3422@ report
51     #1000 ms
52   key? until
53   hex
54 ;
```

Notes on this program:

- `mcp3422-run` is the top-level word that initializes the hardware, then periodically reads the MCP3422 data and reports the voltage (in millivolts) to the user terminal. The program runs until a key is pressed.
- The converted value is read from the MCP3422 as and 18-bit value in 2-complement format. The word `mcp3422@` reads the data as three bytes from the I^2C port and then shuffles it into a double-cell value that is left on the stack, along with a flag to indicate whether the value sent by the MCP3422 happened to be the latest data. If the MCP3422 did not respond to being addressed, zeros will be left on the stack in place of the expected data.
- The value is scaled to microvolts and then the resultant double value is output using the pictured numeric output to have 3 decimal places so that it looks like a millivolt reading. Several lines from the terminal look like the following:

  ```
  new 1028.031 mV
  new 1028.062 mV
  new 1028.046 mV
  ```

- This program builds upon the `i2c-base-k22` words in order to communicate with the MCP3422. The code for scaling of the measured data requires the mixed-scale word `m*/` from the file `math.txt` provided by FlashForth.

13 An I²C slave example

The MSSP in the PIC18F26K22 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 [10]. 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.

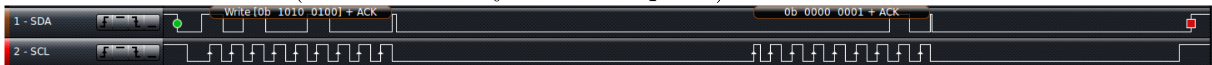
```
1  -i2c-slave
2  marker -i2c-slave
3  \ Make the FlashForth 26K22 demo board into an I2C slave.
4  \ An I2C master can read and write to a buffer here,
5  \ the least-significant bit of the first byte controls
6  \ the LED attached to pin RB0.
7  \
8  \ Needs core.txt loaded.
9
10 $ff81 constant portb
11 $ff82 constant portc
12 $ff8a constant latb
13 $ff93 constant trisb
14 $ff94 constant trisc
15 $ff3a constant anselc
16
17 : led_on ( -- )
18   %00000001 latb mset
19 ;
20 : led_off ( -- )
21   %00000001 latb mclr
22 ;
23 : err_led_on ( -- )
24   %00000010 latb mset
25 ;
26 : err_led_off ( -- )
27   %00000010 latb mclr
28 ;
29
30 \ Establish a couple of buffers in RAM, together with index variables.
31 ram
32 8 constant buflen
33 \ Receive buffer for incoming I2C data.
34 create rbuf buflen allot
35 variable rindx
36 : init_rbuf ( -- )
37   rbuf buflen erase
38   0 rindx !
39 ;
40 : incr_rindx ( -- ) \ increment with wrap-around
41   rindx @ 1 +
42   dup buflen = if drop 0 then
43   rindx !
44 ;
45 : save_to_rbuf ( c -- )
46   rbuf rindx @ + c!
47   incr_rindx
48 ;
49
50 \ Send buffer with something interesting for the I2C master to read.
```

```
51 create sbuf buflen allot
52 variable sindx
53 : incr_sindx ( -- ) \ increment with wrap-around
54   sindx @ 1 +
55   dup buflen = if drop 0 then
56   sindx !
57 ;
58 : init_sbuf ( -- ) \ fill with counting integers, for interest
59   buflen
60   for
61     r@ 1+
62     sbuf r@ + c!
63   next
64   0 sindx !
65 ;
66
67 \ I2C-related definitions and code
68 $ffc5 constant sspcon2
69 $ffc6 constant sspcon1
70 $ffc7 constant sspstat
71 $ffc8 constant sspadd
72 $ffc9 constant sspbuf
73 $ff9e constant pir1
74
75 \ PIR1 bits
76 %00001000 constant sspif
77
78 \ SSPSTAT bits
79 %00000001 constant bf
80 %00000100 constant r_nw
81 %00001000 constant start_bit
82 %00010000 constant stop_bit
83 %00100000 constant d_na
84 %01000000 constant cke
85 %10000000 constant smp
86
87 d_na start_bit or r_nw or bf or constant stat_mask
88
89 \ SSPCON1 bits
90 %00010000 constant ckp
91 %00100000 constant sspen
92 %01000000 constant sspov
93 %10000000 constant wcol
94
95 \ SSPCON2 bits
96 %00000001 constant sen
97
98 : i2c_init ( -- )
99   %11000 anselc mclr \ enable digital-in on RC3,RC4 (SCL1,SDA1)
100  %00011000 trisc mset \ RC3==SCL RC4==SDA
101  %00000110 sspcon1 c! \ Slave mode with 7-bit address
102  sen sspcon2 mset \ Clock stretching enabled
103  smp sspstat mset \ Slew-rate disabled
104  $52 1 lshift sspadd c! \ Slave address
105  sspen sspcon1 mset \ Enable MSSP peripheral
106 ;
107
108 : release_clock ( -- )
109   ckp sspcon1 mset
110 ;
111
112 : i2c_service ( -- )
113   \ Check the state of the I2C peripheral and react.
114   \ See App Note 734 for an explanation of the 5 states.
115   \
116   \ State 1: i2c write operation, last byte was address.
117   \ D_nA=0, S=1, R_nW=0, BF=1
118   sspstat c@ stat_mask and %00001001 =
119   if
120     sspbuf @ drop
121     init_rbuf
```

```
122     release_clock
123     exit
124   then
125   \ State 2: i2c write operation, last byte was data.
126   \ D_nA=1, S=1, R_nW=0, BF=1
127   sspstat c@ stat_mask and %00101001 =
128   if
129     sspbuf c@ save_to_rbuf
130     release_clock
131     exit
132   then
133   \ State 3: i2c read operation, last byte was address.
134   \ D_nA=0, S=1, R_nW=1
135   sspstat c@ %00101100 and %00001100 =
136   if
137     sspbuf c@ drop
138     0 sindx !
139     wcol sspcon1 mclr
140     sbuf sindx @ + c@ sspbuf c!
141     release_clock
142     incr_sindx
143     exit
144   then
145   \ State 4: i2c read operation, last byte was outgoing data.
146   \ D_nA=1, S=1, R_nW=1, BF=0
147   sspstat c@ stat_mask and %00101100 =
148   ckp sspcon1 mtst 0=
149   and
150   if
151     wcol sspcon1 mclr
152     sbuf sindx @ + c@ sspbuf c!
153     release_clock
154     incr_sindx
155     exit
156   then
157   \ State 5: master NACK, slave i2c logic reset.
158   \ From AN734: D_nA=1, S=1, BF=0, CKP=1, however,
159   \ we use just D_nA=1 and CKP=1, ignoring START bit.
160   \ This is because master may have already asserted STOP
161   \ before we service the final NACK on a read operation.
162   d_na sspstat mtst 0 > ckp sspcon1 mtst 0 > and
163   stop_bit sspstat mtst or
164   if
165     exit \ Nothing needs to be done.
166   then
167   \ We shouldn't arrive here...
168   err_led_on
169   cr ." Error "
170   ." sspstat " sspstat c@ u.
171   ." sspcon1 " sspcon1 c@ u.
172   ." sspcon2 " sspcon2 c@ u.
173   cr
174   begin again \ Hang around until watch-dog resets MCU.
175 ;
176
177
178 : init ( -- )
179   %00000011 trisb mclr  \ want RB0,RB1 as output pins
180   init_rbuf
181   init_sbuf
182   i2c_init
183   led_on err_led_on #200 ms led_off err_led_off
184 ;
185
186 : main ( -- )
187   cr ." Start I2C slave "
188   init
189   begin
190     sspif pir1 mtst
191     if
192       sspif pir1 mclr
```

```
193         i2c_service
194       then
195       rbuf c@ %00000001 and
196       if led_on else led_off then
197       cwd
198     key? until
199   ;
200
201   \ ' main is turnkey
```

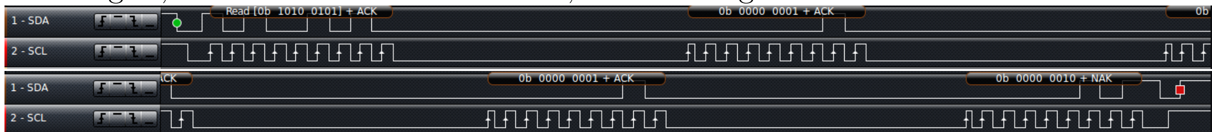
With a Saleae Logic Analyser connected, we can see the I^{2}C signals as a result of writing the byte `0x01` to turn on the LED. The following figure shows the data and clock signals from the time that the master asserts the START condition (green circle) until it asserts the STOP condition (as indicated 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 I^{2}C 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 I^{2}C 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.

14 Speed of operation – bit banging

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 `blink-forth` which toggles an IO pin using the high-level FlashForth words that fetch and store bit patterns into the port latch register. An alternative word `blink-asm` uses assembler instructions to achieve an equivalent effect, but faster, and a third word `blink-bits` uses the FlashForth `bit0:` and `bit1:` words to create high-level bit-manipulation words that also achieve full machine speed.

14.1 PIC18F26K22

```
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.  We also need bit.txt.
6
7 $ff8a constant latb
8 $ff93 constant 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 ;
40
41 \ high-level bit fiddling with named bits
42 latb #1 bit1: RB1-hi inlined
43 latb #1 bit0: RB1-lo inlined
44 : blink-bits ( -- )
45   initRB1
46   begin
47     RB1-hi RB1-lo \ one cycle
48     RB1-hi RB1-lo
49     RB1-hi RB1-lo
50     RB1-hi RB1-lo
51     cwd
```

```
52   again
53 ;
```

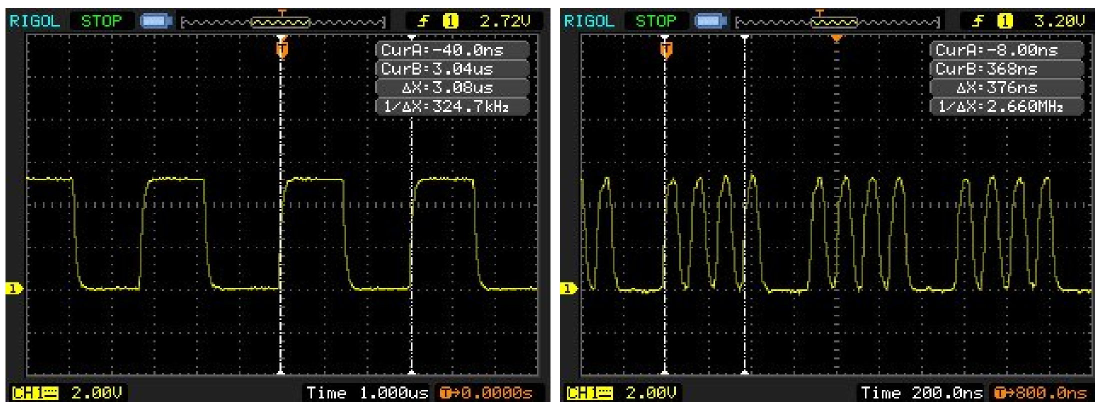
Notes on this program:

- 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 a 31.25 kHz oscillator frequency, this leads to a default timeout period of a little over 1 second ($32\,\mu\text{s} \times 128 \times 256$).

- For the PIC18 MCU, the internal oscillator of 16 MHz was multiplied by the PLL to get 64 MHz oscillator driving the MCU. With 4 clock cycles per instruction cycle, this gave an instruction period $T_{CY} = 62.5\,\text{ns}$. Current consumption by the microcontroller was about 14 mA, roughly double the value when the interpreter is not doing much, just waiting for input.

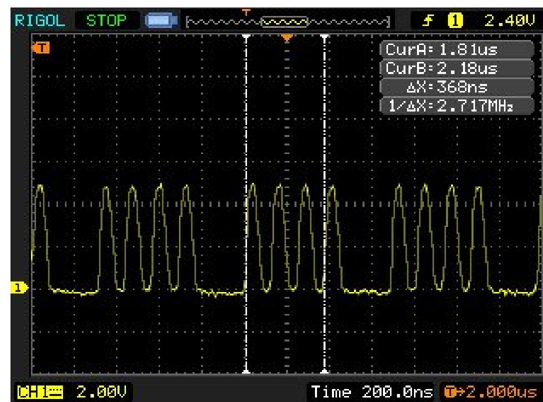
- The screen image on the left shows the output signal for running the high-level `blink-forth` word while the image on the right uses the assembler words.



- For the `blink-forth` word, one on+off cycle of the LED executes in 6 words and is seen (in the oscilloscope record) to require about 50 instruction cycles. So, on average, each of these threaded Forth words is executed in about 8 MCU instruction cycles. 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 125 ns 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.

- The oscilloscope record for the `blink-bits` word is shown here.



With the bit-manipulation words `RB1-hi` and `RB1-lo` being inlined, they also achieve full machine speed because the generated code is essentially the same as for `blink-asm`.

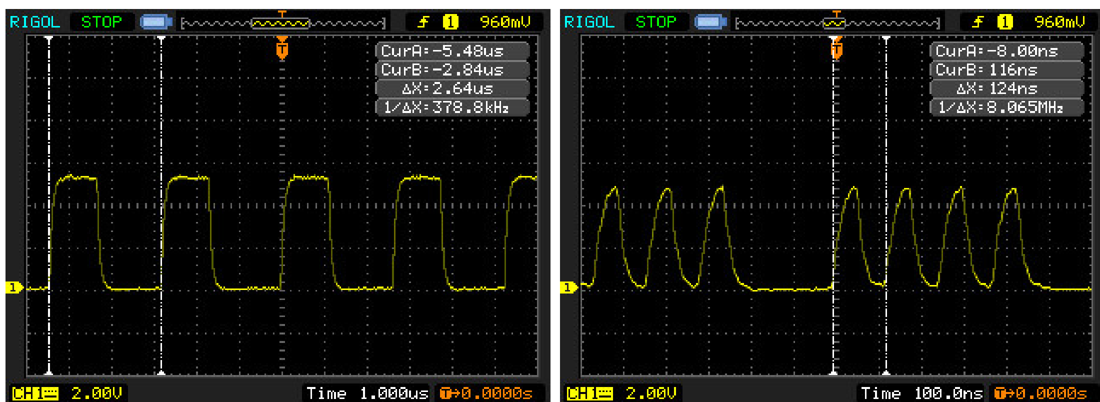
14.2 PIC24FV32KA302

```
1 -speed-test
2 marker -speed-test
3 \ For the PIC24FV32KA302, waggle RB15 as quickly as we can,
4 \ in both high- and low-level code.
5 \ Remember to load bit.txt before this file.
6
7 $02c8 constant trisb
8 $02ca constant portb
9 $02cc constant latb
10 $02ce constant odcb
11
12 1 #15 lshift constant bit15
13
14 : initRB15
15   bit15 trisb mclr \ RB15 as output
16   bit15 latb mclr \ initially known state
17 ;
18
19 \ high-level bit fiddling, presumably slow
20 : blink-forth ( -- )
21   initRB15
22   begin
23     bit15 latb ! 0 latb ! \ one cycle, on and off
24     bit15 latb ! 0 latb !
25     bit15 latb ! 0 latb !
26     bit15 latb ! 0 latb !
27     cwd \ We have to kick the watch dog ourselves.
28   again
29 ;
30
31 \ low-level bit fiddling, via assembler
32 : blink-asm ( -- )
33   initRB15
34   [
35   begin,
36     #15 latb bset,  #15 latb bclr, \ one cycle, on and off
37     #15 latb bset,  #15 latb bclr,
38     #15 latb bset,  #15 latb bclr,
39     #15 latb bset,  #15 latb bclr,
40     ] cwd [ \ kick the watch dog
41   again,
```

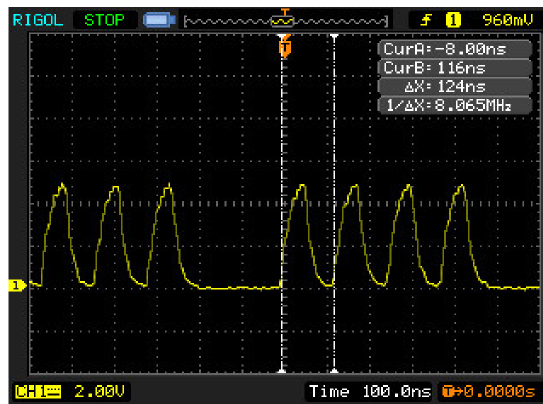
```
42   ]
43 ;
44
45 \ high-level bit fiddling with named bits
46 latb #15 bit1: RB15-hi inlined
47 latb #15 bit0: RB15-lo inlined
48 : blink-bits ( -- )
49   initRB15
50   begin
51     RB15-hi RB15-lo \ one cycle
52     RB15-hi RB15-lo
53     RB15-hi RB15-lo
54     RB15-hi RB15-lo
55     cwd
56   again
57 ;
```

Notes on this program:

- The order of the assembler arguments is bit-number register-address op-code. This is different to that seen in the PIC18 version of the program.
- The MCU was configured for running off its internal 8 MHz oscillator with the 4× PLL active and a 1:1 postscaling. This resulted in an instruction cycle period $T_{CY} = 62.5\,\text{ns}$.
- The screen image on the left shows the output signal for running the high-level `blink-forth` word while the image on the right uses the assembler words.



- For the `blink-forth` word, one on+off cycle of the LED executes in 6 words and is seen (in the oscilloscope record) to require about 42 instruction cycles. So, on average, each of these threaded Forth words is executed by the 16-bit PIC24 in 7 MCU instruction cycles. This illustrates a benefit of the 16-bit processor, since the 8-bit PIC18F26K22 required 50 MCU instruction cycles (and a correspondingly longer time of 3.08 microseconds) for the same effect.
- 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 124 ns period.
- The oscilloscope record for the `blink-bits` word is shown here.



Again, the bit-manipulation words `RB15-hi` and `RB15-lo` also achieve full machine speed.

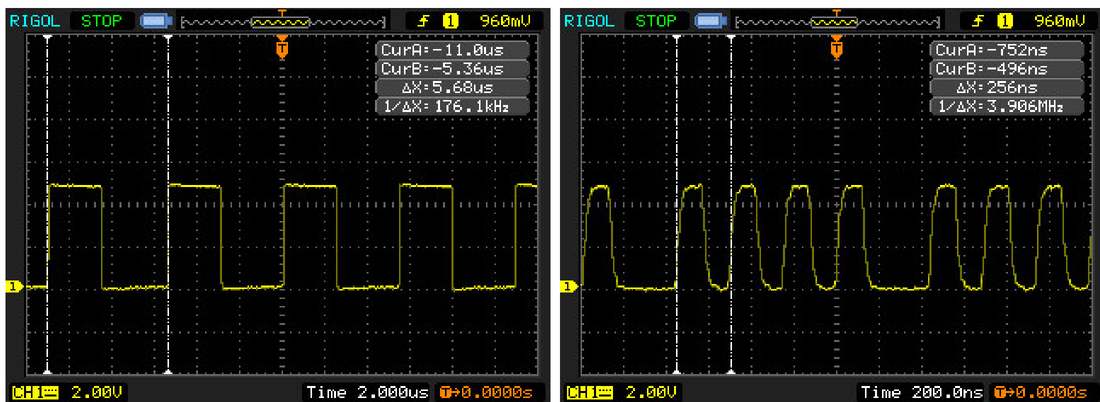
14.3 ATmega328P

```
-speed-test
marker -speed-test
\ Waggle PB5 as quickly as we can, in both high- and low-level code.
\ Before sending this file, we should send asm.txt and bit.txt.

$0024 constant ddrb
$0025 constant portb    \ RAM address
$0005 constant portb-io \ IO-space address
1 #5 lshift constant bit5

: initPB5
  bit5 ddrb mset \ set pin as output
  bit5 portb mclr \ initially known state
;

: cwd ( -- ) [ wdr, ] ; inlined \ we might want to reset the watchdog

\ high-level bit fiddling, presumably slow
: blink-forth ( -- )
  initPB5
  begin
    bit5 portb c! 0 portb c! \ one cycle, on and off
    bit5 portb c! 0 portb c!
    bit5 portb c! 0 portb c!
    bit5 portb c! 0 portb c!
    cwd
  again
;

\ low-level bit fiddling, via assembler
: blink-asm ( -- )
  initPB5
  [
  begin,
    portb-io #5 sbi,  portb-io #5 cbi, \ one cycle, on and off
    portb-io #5 sbi,  portb-io #5 cbi,
    portb-io #5 sbi,  portb-io #5 cbi,
    portb-io #5 sbi,  portb-io #5 cbi,
    wdr,
  again,
  ]
;

\ high-level bit fiddling with named bits
```

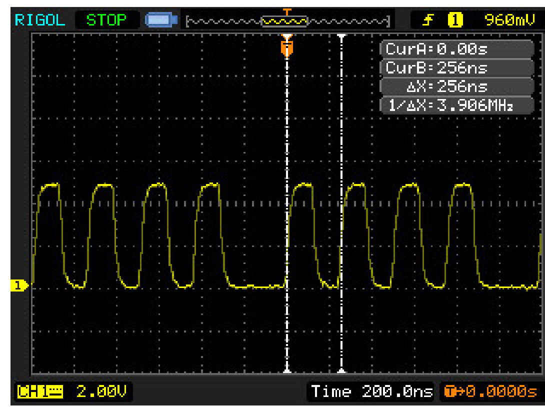
```
45 portb #5 bit1: PB5-hi inlined
46 portb #5 bit0: PB5-lo inlined
47 : blink-bits ( -- )
48   initPB5
49   begin
50     PB5-hi PB5-lo \ one cycle
51     PB5-hi PB5-lo
52     PB5-hi PB5-lo
53     PB5-hi PB5-lo
54     cwd
55   again
56 ;
```

Notes on this program:

- Except for names, this code is essentially the same as for the PIC18 and PIC24 versions of the exercise. FlashForth abstracts away much of the instruction-set architecture of the microcontroller, leaving us to focus on twiddling the bits of the peripheral hardware.
- The MCU was configured for running with the 16 MHz crystal, which resulted in a machine clock cycle period $T_{CY} = 62.5$ ns.
- The screen image on the left shows the output signal for running the high-level `blink-forth` word while the image on the right uses the assembler words.



- For the `blink-forth`, one on+off cycle of the LED executes in 6 words and is seen (in the oscilloscope record) to require about 90 instruction cycles. So, on average, each of these threaded Forth words is executed by the 8-bit AVR in 15 MCU instruction cycles.
- 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 (`sbi` and `cbi`) each requiring 2 clock cycles, so we see a short, quarter-microsecond period.
- The oscilloscope record for the `blink-bits` word is shown here.



It can be seen that the bit-manipulation words `PB5-hi` and `PB5-lo` achieve full machine speed.

15 Driving an Hitachi-44780 LCD controller

The LCD in the photograph on page 7 was driven with the following code. During the development of this example, a lesson was relearned – that of reading the data sheet [11] carefully :)

```
1 \ Exercise LCD on PICDEM2+ board.
2 \ Remember to load bit.txt before this file.
3 -xlcd
4 marker -xlcd
5
6 $ff80 constant porta
7 $ff89 constant lata
8 $ff92 constant trisa
9 $ff83 constant portd
10 $ff8c constant latd
11 $ff95 constant trisd
12
13 \ The LCD is operated in nibble mode.
14 \ RA1 = Enable (E) pin
15 \ RA2 = Read/Write (RW) pin
16 \ RA3 = Register Select (RS) pin
17 \ RD0 = DB4 on LCD
18 \ RD1 = DB5
19 \ RD2 = DB6
20 \ RD3 = DB7
21
22 portd constant dataport
23 lata #1 bit0: Elo
24 lata #1 bit1: Ehi
25 lata #2 bit0: RWlo
26 lata #2 bit1: RWhi
27 lata #3 bit0: RSlo
28 lata #3 bit1: RShi
29
30 : data-port-in ( -- )
31   trisd c@ $0f or trisd c!
32 ;
33
34 : data-port-out ( -- )
35   trisd c@ $f0 and trisd c!
36 ;
37
38 : put-nibble ( c -- )
39   \ Make lower 4 bits of c appear on data port pins.
40   $0f and
41   dataport c@ $f0 and
42   or
43   dataport c!
44 ;
45
46 : short-delay ( -- )
47   18 for r@ drop next ;
48
49 : Estrobe ( -- )
50   Ehi short-delay Elo
51 ;
52
53 : lcd-getc ( -- c )
54   \ Read the LCD register in two nibbles.
55   \ Remember to select the register line before calling this word.
56   data-port-in
57   RWhi short-delay
58   Ehi short-delay dataport c@ #4 lshift Elo short-delay \ high nibble
59   Ehi short-delay dataport c@ Elo short-delay \ low nibble
60   or \ assemble full byte and leave it on the stack
61   RWlo short-delay
62 ;
```

```
63
64 : lcd-ready? ( -- f )
65   \ Read the command register and check busy bit.
66   RSlo short-delay
67   lcd-getc $80 and 0=
68 ;
69
70 : wait-for-lcd ( -- )
71   begin lcd-ready? cwd until
72 ;
73
74 : lcd-putc ( c -- )
75   \ Write the LCD register in two nibbles.
76   \ Remember to select the register line before calling this word.
77   dup $f0 and #4 rshift \ high nibble left on top of stack
78   data-port-out
79   RWlo short-delay
80   put-nibble short-delay Estrobe short-delay
81   $0f and \ low nibble now left on top of stack
82   put-nibble short-delay Estrobe short-delay
83   data-port-in
84 ;
85
86 : lcd-clear ( -- )
87   wait-for-lcd
88   RSlo short-delay
89   %00000001 lcd-putc
90 ;
91
92 : lcd-home ( -- )
93   wait-for-lcd
94   RSlo short-delay
95   %00000010 lcd-putc
96 ;
97
98 : lcd-goto ( c -- )
99   \ Set the specified 7-bit data memory address.
100  wait-for-lcd
101  RSlo short-delay
102  $80 or \ sets the highest bit for the command
103  lcd-putc
104 ;
105
106 : lcd-init ( -- )
107  data-port-in
108  Elo RWlo RSlo
109  %00001110 trisa mclr \ RS, RW and E as output
110  30 ms \ power-on delay
111  \ Begin "initialization by instruction"
112  \ Presumably, the LCD is in 8-bit interface mode.
113  %0011 put-nibble Estrobe 5 ms
114  %0011 put-nibble Estrobe 1 ms
115  %0011 put-nibble Estrobe 1 ms
116  \ Function set for 4-bit interface; it is still in 8-bit mode.
117  %0010 put-nibble Estrobe 1 ms
118  \ Now, we should be in 4-bit interface mode.
119  \ Function set for 4-bit interface, 2 display lines 5x7 font.
120  wait-for-lcd
121  %00101000 lcd-putc
122  \ Increment cursor after each byte, don't shift display.
123  wait-for-lcd
124  %00000110 lcd-putc
125  \ Display off
126  wait-for-lcd
127  %00001000 lcd-putc
128  \ Display clear
129  %00000001 lcd-putc
130  5 ms
131  \ End of "initialization by instruction"
132  \ Enable cursor and display, no blink.
133  wait-for-lcd
```

```
134   %00001110 lcd-putc 1 ms
135   wait-for-lcd
136 ;
137
138 : lcd-emit ( c -- ) \ Write the byte into data memory.
139   wait-for-lcd
140   RShi short-delay
141   lcd-putc
142 ;
143
144 : lcd-type ( c-addr n -- ) \ send string
145   for c@+ lcd-emit next
146   drop
147 ;
148
149 : main
150   ." Begin..."
151   lcd-init
152   cr ." lcd-init done."
153   s" Hello from" lcd-type
154   $40 lcd-goto
155   s" FlashForth 5.0" lcd-type
156   cr ." exercise done."
157 ;
```

References

[1] Mikael Nordman. FLASHFORTH: for PIC and Atmega. URL http://flashforth.com, mikael.nordman@flashforth.com, 2014.

[2] Microchip Technology Inc. PIC18(L)F2X/4XK22 data sheet 28/40/44-pin, low-power, high-performance microcontrollers with XLP technology. Datasheet DS41412F, Microchip Technology Inc., www.microchip.com, 2012.

[3] Microchip Technology Inc. PIC24FV32KA304 FAMILY 20/28/44/48-pin, general purpose, 16-bit flash microcontrollers with XLP technology. Datasheet DS39995D, Microchip Technology Inc., www.microchip.com, 2013.

[4] Atmel. Atmel 8-bit Microcontroller with 4/8/16/32KBytes In-System Programmable Flash ATmega48A; ATmega48PA; ATmega88A; ATmega88PA; ATmega168A; ATmega168PA; ATmega328; ATmega328P. Datasheet 8271GAVR02/2013, www.atmel.com, 2013.

[5] 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.

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

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

[8] Peter Jacobs, Peter Zawasky, and Mikael Nordman. Elements of FlashForth. School of Mechanical and Mining Engineering Technical Report 2013/08, The University of Queensland, Brisbane, May 2013.

[9] Microchip Technology Inc. MCP3422/3/4: 18-bit, multi-channel $\Delta\Sigma$ analog-to-digital converter with I2C interface and on-board reference. Datasheet DS22088C, Microchip Technology Inc., www.microchip.com, 2009.

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

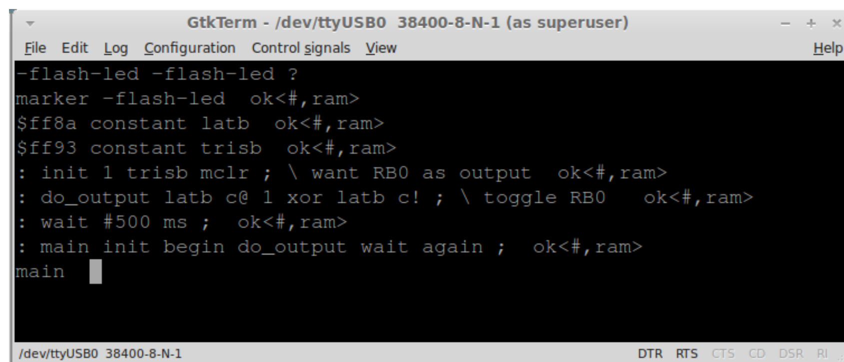
[11] Hitachi. HD44780U (LCD-II) (Dot Matrix Liquid Crystal Display Controller/Driver). Datasheet ADE-207-272(Z) '99.9 Rev. 0.0, Hitachi., 1999.

A Using other terminal programs on Linux

As discussed in Section 4, interaction with the programmed MCU is via the serial port. There are a number of generic terminal emulation programs that will communicate via a serial port. 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. `GtkTerm` is available as more conventional terminal program. The following images shows the GtkTerm window just after sending the content of the `flash-led.txt` file to the PIC18F26K22. The device name of `/dev/ttyUSB0` refers to the USB-to-serial interface that was plugged one of the PC's USB ports. It is convenient to start GtkTerm with the command

```
$ sudo gtkterm
```

and then adjust the communication settings via the `Configuration` → `Port` menu item and its associated dialog window.



There is also a send-file capability and, importantly, the capability to set the period between lines of text 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, an end-of-line delay of a few tens of milliseconds has usually been found adequate, however, there have been times that a file would not successfully load until the end-of-line pause was increased to 300 milliseconds. For `GtkTerm`, this setting is under the `Advanced Configuration Options` in the port configuration dialog, as shown below. This end-of-line delay makes the transfer of large files slow, however, the text still scrolls past quickly but is now at a pace where it is possible to follow the dialog and know how well the compilation is going. Building your application code incrementally, with small files, is a good thing.

Configuration (as superuser)

Serial port

Port:	Baud Rate:	Parity:
/dev/ttyUSB0	38400	none
Bits:	Stopbits:	Flow control:
8	1	Xon/Xoff

Advanced Configuration Options

ASCII file transfer

End of line delay (milliseconds): 300

Wait for this special character before passing to next line:

RS485 half-duplex parameters (RTS signal used to send)

Time with RTS 'on' before transmit (milliseconds): 30

Time with RTS 'on' after transmit (milliseconds): 30

OK Cancel