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

Mechanical Engineering Report 2014/02
P. A. Jacobs
School of Mechanical and Mining Engineering
The University of Queensland.

May 20, 2014

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 PIC24FV16KM202 and AVR ATmega328P microcontrollers is also covered.
## Contents

1. **A selection of microcontrollers**  
2. **Development boards**  
   2.1 PIC18 family boards  
   2.2 AVR and PIC24 boards  
3. **FlashForth**  
   3.1 Getting FlashForth and programming the MCU  
   3.2 Building for the PIC18F26K22 or PIC18F46K22  
   3.3 Building for the PIC24FV16KM202  
   3.4 Building for the ATmega328P  
4. **Interacting with FlashForth**  
5. **Introductory examples**  
   5.1 Hello, World: Flash a LED on the PIC18F26K22  
   5.2 Flash a LED on the PIC24FV16KM202  
   5.3 Flash a LED on the ATmega328P  
   5.4 Set the cycle duration with a variable (PIC18F26K22)  
   5.5 Hello, World: Morse code  
6. **Read and report an analog voltage**  
7. **Counting button presses**  
8. **Counting button presses via interrupts**  
9. **Scanning a 4x3 matrix keypad**  
10. **Using I2C to get temperature measurements**  
11. **Making high-resolution voltage measurements**  
12. **An I2C slave example**  
13. **Speed of operation**  
   13.1 PIC18F26K22  
   13.2 PIC24FV16KM202  
   13.3 ATmega328P  
14. **Driving an Hitachi-44780 LCD controller**
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 PIC24FV16KM202 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 interface 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

Extremely 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 – PIC18FXXX22 devices
- 1.8V to 3.6V Operation – PIC18LFXXXX22 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-Reset
  - PWM steering
- Two Master Synchronous Serial Port (MSSP) modules:
  - 3-wire SPI (supports all 4 modes)
  - I²C™ Master and Slave modes with address mask

© 2010-2012 Microchip Technology Inc. DS41412F-page 3
**FIGURE 1-1: PIC18(L)F2X/4XK22 FAMILY BLOCK DIAGRAM**

- **Internal Oscillator Block**
  - LFINTOS Oscillator
  - 16 MHz Oscillator
- **Power-up Timer**
- **Watchdog Timer**
- **Brown-out Reset**
- **Fail-Safe Clock Monitor**
- **Comparator**
- **ADC**
- **DAC**

**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 2.0.9 “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.
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 14. 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 I²C 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.
pic18f26k22 not–quite minimal demo board

Peter Jacobs

30–Apr–2014
### 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 PIC24FV16KM202, 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

FILE: REVISION: DRAWN BY: PAGE OF
TITLE
2
4
6
1
3
5
CONN_ICSP
+5V
1N4004
1k
1k
AVR ATmega328 not−quite minimal demo board
Peter Jacobs
2 DEVELOPMENT BOARDS

PIC24FV16KM202 not-quite minimal demo board

Peter Jacobs

30-Apr-2014
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.

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. To make the build process a little easier, a set of MPLAB-X projects is provided in the PIC18 directory of the FlashForth source distribution. We select the FF_UART.X project to build our FlashForth program that will use the microcontroller’s UART serial port as the OPERATOR communications channel.

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 $F_{OSC} = 64$ MHz), enable the watchdog timer with a 1:32768 postscale ($WDTPS = 32768$), 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 needed to edit the pic18f-main.cfg file only to set the system clock speed as constant clock=d’64000000’. 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 423 bytes of SRAM used (another 3473 free) and 8808 bytes of program memory used (56728 free), for the PIC18F26K22 MCU, FlashForth occupies only about one-eighth of the microcontroller’s memory. The rest 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 selected to use of the MPLAB ICD3.
3.3 Building for the PIC24FV16KM202

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 `p24f16ka_config.inc` file for our PIC24FV16KM202-I/SP microcontroller using its internal 8 MHz oscillator and installed on the 5V Microstick starter board:

```forth
ifdef __24FV16KM202
.print "Customize p24f16ka_config.inc for PIC24FV16KM202"

.equ FLASH_SIZE, 0x2C00 ; 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

cfg FOSCSEL, FNOSC_FRCDIV & SOSCSRC_ANA & LPRCSEL_LP & IESO_OFF

cfg FOSC, POSCMOD_NONE & OSCIOFNC_IO & POSCFREQ_MS & SOSCSEL_SOSCLP & FCKSM_CSDCMD

cfg FWDT, FWDTEN_ON

cfg FSROR, BOREN_BOR3 & RETCFG_OFF & PWREN_OFF & I2C1SEL_PRI & BORV_V18 & MCLRE_ON
; ICS_PGx2 for strip-board example
; ICS_PGx1 for 5V Microstick for PIC24 K-series

cfg FOCM, ICS_PGx1

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

.endif
```

Once programmed, FlashForth uses 646 of the microcontroller’s 2048 bytes of SRAM and 4431 of the MCU’s 5632 words of Flash memory. This leaves 1402 bytes of SRAM and 1201 words of Flash program 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 and 1201 words can hold a lot of Forth instructions.

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.

```bash
$ sudo avrdude -p m328p -B 8.0 -c jtag3isp -P usb -e \
  -U efuse:w:0xff:m \n  -U hfuse:w:0xda:m \n  -U lfuse:w:0xff:m \n  -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. Settings are 38400 baud 8-bit, no parity, 1 stop bit, with software (Xon/Xoff) flow control.
On a Linux machine, the cutecom terminal program is very convenient. It has a line-oriented input that doesn’t send the text to the MCU until you press the enter key. This allows for editing of the line before committing it to the MCU and convenient recall of previous lines. GtkTerm is available as a more conventional terminal program. The following images show 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 into 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.

![(GtkTerm window)](image)

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.

![(Advanced Configuration Options)](image)
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 PIC18F26K22

The microcontroller version of the “Hello, World” program is typically a program that flashes a single LED. It makes use of a digital input-output pin via the registers that control the IO port. The datasheet [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.
5 INTRODUCTORY EXAMPLES

- Line 7 defines a word to pause for 500 milliseconds.
- 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 kill the application and put the MCU back into a state of listening to the serial port. Typing main, followed by Enter will restart the application.

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

```forth
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 PIC24FV16KM202

```forth
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

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

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 (PIC18F26K22)

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.

```fortran
- flash-led-var
marker - flash-led-var
\ Flash a LED attached to pin RB0.
$ff8a constant latb
$ff93 constant trisb
variable ms_count \ use this for setting wait period.
: init ( -- )
1 trisb mclr \ want RB0 as output
;
: do_output ( -- )
latb c@ 1 xor latb c! \ toggle RB0
;
: wait ( -- )
ms_count @ ms
;
: main ( n -- )
ms_count ! \ store for later use in wait
init
```
5.5 Hello, World: Morse code

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

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

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

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

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

\ Have looked up the ARRL CW list for the following letters.
H dit dit dit dit ;
e dit ;
l dit dit ;
o dah dah dah ;
```
6 Read and report an analog voltage

Use of the analog-to-digital converter (ADC) is a matter of, first, reading Section 17 of the PIC18F26K22 datasheet, setting the relevant configuration/control registers and then giving it a poke when we want a measurement. Again, the interactive nature of FlashForth makes the reporting of the measured data almost trivial.
7 COUNTING BUTTON PRESSES

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

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.
COUNTING BUTTON PRESSES VIA INTERRUPTS

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.

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.

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

main \ exercise the application
```
COUNTING BUTTON PRESSES VIA INTERRUPTS

8  $fff1 constant intcon2
9  variable count
10  variable last-count
11
12  : int0-irq
13     [i
14     10 intcon mset \ INTOIF
15     if
16     count @ 1+ count !
17     10 intcon mclr
18     then
19     i]
20
21  : init ( -- )
22     01 trisb mset \ RB0 as input, a button press will give 0.
23     01000000 intcon2 mclr \ interrupt on falling edge
24     '[' int0-irq 0 int \ install service word
25     10 intcon mclr \ INTOIF cleared
26     10000 intcon mset \ INTO interrupt enable
27
28  : main ( -- )
29     0 count !
30     init
31     begin
32     count @ last-count @ = \ change?
33     if
34     count @ dup last-count ! .
35     then
36     cwd
37     key? until
38     ;
39
40  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 INTO 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 INTOIF 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.

```assembly
: init ( -- )
0 latb c!
%00000000 anselb c! \ set as all digital I/O pins
%11110000 trisb c! \ RB7-4 as input, RB3-0 as output
%11110000 wpub c! \ pull-ups on RB7-4
%10000000 intcon2 mclr \ turn on pull-ups
;
flash
create key_chars
char 1 c, char 2 c, char 3 c,
char 4 c, char 5 c, char 6 c,
char 7 c, char 8 c, char 9 c,
char * c, char 0 c, char # c,
create key_scan_bytes
$7e c, $7d c, $7b c,
$be c, $bd c, $bb c,
$de c, $dd c, $db c,
$ee c, $ed c, $eb c,
ram

: scan_keys ( -- c )
\ Return ASCII code of key that is pressed
#12 for
key_scan_bytes r0 + c0
dup
latb c!
portb c0
= if
\ key must be pressed to get a match
key_chars r0 + c0
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  \ 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 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 Using I2C to get temperature measurements

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

\ Read temperature from TC74 on PICDEM2+ board with PIC18F46K22-I/P.
\ Modelled on Mikael Nordman's i2c_tcn75.txt.
\ This program requires i2c_base.txt to be previously loaded.
-read-tc74
5 marker -read-tc74
6 $ff3a constant ansc
7 %1001101 constant addr-tc74 \ default 7-bit address for TC74
8 add-read-bit ( 7-bit-c -- 8-bit-c )
9 \ Make 8-bit i2c address with bit 0 set.
10 1 lshift 1 or
11 ;
12 add-write-bit ( 7-bit-c -- 8-bit-c )
13 \ Make 8-bit i2c address with bit 0 clear.
14 1 lshift 1 invert and
15 ;
16 sign-extend ( c -- n )
17 \ If the TC74 has returned a negative 8-bit value,
18 \ we need to sign extend to 16-bits with ones.
19 dup $7f > if $ff80 or then
20 ;
21 init-tc74 ( -- )
22 \ Selects the temperature register for subsequent reads.
23 addr-tc74 add-write-bit i2cws 0 i2c ! spen
24 ;
25 degrees@ ( -- n )
26 \ Wake the TC74 and receive its register value.
27 addr-tc74 add-read-bit i2cws i2c@nak sign-extend
28 ;
29 main ( -- )
30 %11000 anselc mclr \ enable digital-in on RC3,RC4 (SCL1,SDA1)
31 i2cinit
32 init-tc74
33 begin
34 degrees@ .
35 #1000 ms
36 key? until
37 ;
38 \ Now, report temperature in degrees C
39 \ while we warm up the TC74 chip with our fingers...
40 decimal main

With a Saleae Logic Analyser connected to the pins of the TC74A5, we can see the I^2C signals as a result of calling the init-tc74 word.

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

Notes on this program:

- This builds upon the i2c_base and asm words supplied with FlashForth as separate text files in the source distribution. You will need to send asm.txt to the microcontroller, followed by i2c_base.txt, in order to have the words i2cws, i2c!, spen and i2c@nak available.
11 Making high-resolution voltage measurements

The Microchip MCP3422 is a ΣΔ-ADC that can connected via I^2C 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^2C 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.

\begin{verbatim}
1 \ mcp3422.txt
2 \ Play with mcp3422 eval board.
3 \ PJ, 21-Oct-2013
4 \ 28-Apr-2014 PIC18F26K22 version
5 \ Needs i2c_base.txt and math.txt.
6
7 -mcp3422
8 marker -mcp3422
9
10 $ff3a constant anselc
11
12 : mcp3422init ( -- )
13 \ $d0 is default mcp4322 address for writing
14 \ $9c is config for 18-bit continuous conversions of ch 1
15 \ $d0 i2cws $9c i2c ! spen
16 ;
17
18 : mcp3422@ ( -- d f ) \ Read the 18-bit result as 3 bytes
19 \ $d1 i2cws i2c@ak \ only 2 bits in first byte
20 dup $3 > if $fffa or then \ sign-extend to full cell
21 i2c@ak $8 lshift i2c@ak or \ next two bytes into one cell
22 swap \ leave double result
23 i2c@nak $80 and 0= \ leave true if result is latest
24 ;
25
26 : microvolts ( d1 -- d2 ) \ The least-significant bit corresponds to 15.625 microvolts
\end{verbatim}
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 I²C 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.

- 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** and **asm** words supplied with FlashForth, in order to communicate with the MCP3422. The code for scaling of the measured data requires words from **math.txt**.
12 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 \ Make the FlashForth 26K22 demo board into an I2C slave.
2 \ An I2C master can read and write to a buffer here,
3 \ the least-significant bit of the first byte controls
4 \ the LED attached to pin RB0.
5 \ Needs core.txt loaded.
6
7 00000001 constant latb
8 00000001 constant mset
9 00000010 constant mclr
10
11 \ Establish a couple of buffers in RAM, together with index variables.
12 ram
13 8 constant buflen
14 \ Receive buffer for incoming I2C data.
15 create rbuf buflen allot
16 variable rindx
17 : init_rbuf ( -- )
18 rbuf buflen erase
19 0 rindx !
20
21 : incr_rindx ( -- ) \ increment with wrap-around
22 rindx @ 1 +
23 dup buflen = if drop 0 then
24 rindx !
25
26 : save_to_rbuf ( c -- )
27 rbuf rindx @ + c!
28 incr_rindx
29
30 \ Send buffer with something interesting for the I2C master to read.
create sbuf buflen allot
variable sindx

: incr_sindx ( -- ) \ increment with wrap-around
sindx @ 1 +
dup buflen = if drop 0 then
sindx !
;
: init_sbuf ( -- ) \ fill with counting integers, for interest
buflen
for
r@ 1+
sbuf r@ + c!
next
0 sindx !
;
\ I2C-related definitions and code
$ffc5 constant sspcon2
$ffc6 constant sspcon1
$ffc7 constant sspsfrat
$ffc8 constant sspsfradd
$ffc9 constant sspsfrbuf
$ff9e constant piri

\ PIR1 bits
%00001000 constant sspsfrif

\ SSPSTAT bits
%00000001 constant bf
%00000100 constant r_nw
%00001000 constant start_bit
%00010000 constant stop_bit
%01000000 constant d_na
%10000000 constant cke
%10000000 constant smp

d_na start_bit or r_nw or bf or constant stat_mask
\ SSPCON1 bits
%00100000 constant ckp
%01000000 constant sspsfron
%10000000 constant sspsfror
%10000000 constant wcol
\ SSPCON2 bits
%00000001 constant sen
%00000001 constant sen

: i2c_init ( -- )
%11000 anselc mclr \ enable digital-in on RC3,RC4 (SCL1,SDA1)
%00011000 trisc mset \ RC3==SCL RC4==SDA
%00001000 trisc mset \ RC3==SCL RC4==SDA
%00000110 sspsfron c! \ Slave mode with 7-bit address
sem sspsfron2 mset \ Clock stretching enabled
sm p sspsfrat mset \ Slew-rate disabled
$52 1 lshift sspsfradd c! \ Slave address
sspsfr con sspsfron mset \ Enable MSSP peripheral
;
: release_clock ( -- )
ckp sspsfron mset
;
: i2c_service ( -- )
\ Check the state of the I2C peripheral and react.
\ See App Note 734 for an explanation of the 5 states.
\ State 1: i2c write operation, last byte was address.
\ D_nA=0, S=1, R_nW=0, BF=1
sspsfrat c@ stat_mask and %00001001 =
if
sspsfrbuf @ drop
init_rbuf
release_clock
exit
then
\ State 2: i2c write operation, last byte was data.
\ D_nA=1, S=1, R_nW=0, BF=1
sspstat c@ stat_mask and %00101001 =
if
ssbuf c@ save_to_rbuf
release_clock
exit
then
\ State 3: i2c read operation, last byte was address.
\ D_nA=0, S=1, R_nW=1
sspstat c@ %00101100 and %00001100 =
if

then
\ State 4: i2c read operation, last byte was outgoing data.
\ D_nA=1, S=1, R_nW=1, BF=0
sspstat c@ stat_mask and %00101100 =
ckp sspcon1 mtst 0=
and
if

then
\ State 5: master NACK, slave i2c logic reset.
\ From AN734: D_nA=1, S=1, BF=0, CKP=1, however,
\ we use just D_nA=1 and CKP=1, ignoring START bit.
\ This is because master may have already asserted STOP
\ before we service the final NACK on a read operation.
d_na sspstat mtst 0 > ckp sspcon1 mtst 0 > and
stop_bit sspstat mtst or
if
exit \ Nothing needs to be done.
then
\ We shouldn't arrive here...
err_led_on
cr ." Error "
." sspstat " sspstat c@ u.
." sspcon1 " sspcon1 c@ u.
." sspcon2 " sspcon2 c@ u.
cr
begin again \ Hang around until watch-dog resets MCU.

:\ init ( -- )
%00000011 trisb mclr \ want RB0,RB1 as output pins
init_rbuf
init_sbuf
i2c_init
led_on err_led_on #200 ms led_off err_led_off

:\ main ( -- )
.cr ." Start I2C slave "
init
begin
sspif pirl mtst
if
sspif pirl mclr
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 \(\mu\)s 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 \(\mu\)s resulting in about 200 \(\mu\)s 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 \(\mu\)s 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 \(\mu\)s period needed by the slave firmware to service the associated SSPIF event.
13 Speed of operation

All of this nice interaction and convenience has some costs. One cost is the number of MCU instruction cycles needed to process the Forth words. To visualize this cost, the following program defines a word `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.

13.1 PIC18F26K22

```forth
- speed - test
marker - speed - test
\ Waggle RB1 as quickly as we can, in both high- and low-level code.
\ Before sending this file, we should send asm . txt so that we have
\ the clrwdt , word available. We also need bit . txt.

$ff8a constant latb
$ff93 constant trisb

: initRB1
%10 trisb mclr \ RB1 as output
%10 latb mclr \ initially known state
;

\ high-level bit fiddling, presumably slow
: blink-forth ( -- )
initRB1
begin
%10 latb c! 0 latb c! \ one cycle, on and off
%10 latb c! 0 latb c!
%10 latb c! 0 latb c!
%10 latb c! 0 latb c!
cwd \ We have to kick the watch dog ourselves.
again
;

\ low-level bit fiddling, via assembler
: blink-asm ( -- )
initRB1
[
begin,
latb 1 a, bsf , latb 1 a, bcf , \ one cycle, on and off
latb 1 a, bsf , latb 1 a, bcf ,
latb 1 a, bsf , latb 1 a, bcf ,
latb 1 a, bsf , latb 1 a, bcf ,
clrwdt , \ kick the watch dog
again,
]
;

\ high-level bit fiddling with named bits
latb #1 bit1: RB1-hi inlined
latb #1 bit0: RB1-lo inlined
: blink-bits ( -- )
initRB1
begin
RB1-hi RB1-lo \ one cycle
RB1-hi RB1-lo
RB1-hi RB1-lo
RB1-hi RB1-lo
cwd
```
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 32768. With a 31.25 kHz oscillator frequency, this leads to a default timeout period of a little over 134 seconds ($32 \mu s \times 128 \times 32768$).

- 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 13 mA.

- 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 newly created 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.

13.2 PIC24FV16KM202

```plaintext
1  -speed-test
2  marker -speed-test
3 \ For the PIC24FV16KM202, 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 #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,
```
13 SPEED OF OPERATION

```plaintext
13 SPEED OF OPERATION

\[ \text{cwd [ \ kick the watch dog again, ]} \]
\[ \text{high-level bit fiddling with named bits} \]
\[ \text{latb #15 bit1: RB15-hi inlined} \]
\[ \text{latb #15 bit0: RB15-lo inlined} \]
\[ \text{: blink-bits ( -- )} \]
\[ \text{initRB15} \]
\[ \text{begin} \]
\[ \text{RB15-hi RB15-lo \ one cycle} \]
\[ \text{RB15-hi RB15-lo} \]
\[ \text{RB15-hi RB15-lo} \]
\[ \text{RB15-hi RB15-lo} \]
\[ \text{cwd} \]
\[ \text{again} \]
\[ ; \]
```

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 a 1:1 postscaling. This resulted in an instruction cycle period \( T_{CY} = 250 \text{ns} \). Although this is 4 times longer than for the PIC18, the current consumption was correspondingly lower at about 3 mA.

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

- For the \textbf{blink-forth} word, one on+off cycle of the LED executes in 6 words and is seen (in the oscilloscope record) to require about 41 instruction cycles. So, on average, each of these threaded Forth words is executed by the 16-bit PIC24 in about 7 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 so we see a short 500 ns period.

- The oscilloscope record for the \textbf{blink-bits} word is shown here.
Again, the newly created bit-manipulation words \texttt{RB15-hi} and \texttt{RB15-lo} also achieve full machine speed.

### 13.3 ATmega328P

```plaintext
- 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, bit.txt and bio.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
drbit 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!
wdr
again
;

\ low-level bit fiddling, via assembler
: blink-assemble ( -- )
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
```
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 \texttt{blink-forth} word while the image on the right uses the assembler words.

- For the \texttt{blink-forth}, one on+off cycle of the LED executes in 6 words and is seen (in the oscilloscope record) to require about 87 instruction cycles. So, on average, each of these threaded Forth words is executed by the 16-bit PIC24 in about 14.5 MCU clock 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 250 ns period.

• The oscilloscope records for the blink-bits and blink-bits-io words are shown here.

![Oscilloscope images](image_url)

The left screen image shows the pin signal when accessing the bit via memory-space address. Here the pulse width was about 620 ns so each of the 2 words required about 5 MCU clock cycles. Although faster than the threaded words, this is not quite the full machine speed which is available when accessing the bit via IO-space address (right screen image).
Driving an Hitachi-44780 LCD controller

The LCD in the photograph on page 6 was driven with the following code. During the development of this example, a lesson was relearned – that of reading the data sheet [11] carefully :)
: lcd-ready? ( -- f )
\ Read the command register and check busy bit.
RSlo short-delay
lcd-getc $80 and 0-
;
:
wait-for-lcd ( -- )
begin lcd-ready? cwd until
;
:
lcd-putc ( c -- )
\ Write the LCD register in two nibbles.
\ Remember to select the register line before calling this word.
dup $f0 and #4 rshift \ high nibble left on top of stack
data-port-out
RWlo short-delay
put-nibble short-delay Estrobe short-delay
$0f and \ low nibble now left on top of stack
put-nibble short-delay Estrobe short-delay
data-port-in
;
:
lcd-clear ( -- )
wait-for-lcd
RSlo short-delay
%00000001 lcd-putc
;
:
lcd-home ( -- )
wait-for-lcd
RSlo short-delay
%00000010 lcd-putc
;
:
lcd-goto ( c -- )
\ Set the specified 7-bit data memory address.
wait-for-lcd
RSlo short-delay
$80 or \ sets the highest bit for the command
cd-putc
;
:
lcd-init ( -- )
data-port-in
Elo RWlo RSlo
%00001110 trisa mclr \ RS, RW and E as output
30 ms \ power-on delay
\ Begin "initialization by instruction"
\ Presumably, the LCD is in 8-bit interface mode.
%0011 put-nibble Estrobe 5 ms
%0011 put-nibble Estrobe 1 ms
%0011 put-nibble Estrobe 1 ms
\ Function set for 4-bit interface; it is still in 8-bit mode.
%0010 put-nibble Estrobe 1 ms
\ Now, we should be in 4-bit interface mode.
\ Function set for 4-bit interface, 2 display lines 5x7 font.
wait-for-lcd
%00100001 lcd-putc
\ Increment cursor after each byte, don't shift display.
wait-for-lcd
%00001110 lcd-putc
\ Display off
wait-for-lcd
%00001000 lcd-putc
\ Display clear
%00000001 lcd-putc
5 ms
\ End of "initialization by instruction"
\ Enable cursor and display, no blink.
wait-for-lcd
\%00001110 lcd-putc 1 ms
wait-for-lcd
;

: lcd-emit ( c -- ) \ Write the byte into data memory.
  wait-for-lcd
  RShi short-delay
  lcd-putc
  ;

: lcd-type ( c-addr n -- ) \ send string
  for c@+ lcd-emit next
  drop
  ;

: main
  ." Begin..."
  lcd-init
  ." lcd-init done."
  s" Hello from" lcd-type
  $40 lcd-goto
  s" FlashForth 5.0" lcd-type
  ." exercise done."
  ;
References


