MyForth

REFERENCE MANUAL

Revision 3.7
November 17, 2006

Prepared By: Bob Nash (Bob.Nash1@Gmail.com)

Document: MyForth Reference Manual.doc

Contents

1

Introduction

Purpose

This manual provides general and technical information for MyForth, an 8-bit Forth for 8051 family processors. MyForth is hosted by GForth and thus can run in both Windows and Linux environments.

MyForth was written by Charles Shattuck and is based on his many years experience in programming in Forth on 8051 processors, primarily while working at AM Research. Although the amrForth system provides a very robust and full-featured 16-bit Forth system for 8051 development, Charley designed MyForth to explore and apply several of his ideas about 8051development in Forth. This exploration was not possible within the context of a mature Forth system like amrForth. One of the reasons for departing from the 16-bit Forth model used by AMR Forth is Charley's conviction that an 8-bit model is appropriate for an 8-bit machine. Although 16, 32 and larger operations are need for tasks such as scaling, these can be considered as special cases to be coded as needed. Mostly, 8051 programming deals with 8-bit operations.

Another reason that MyForth has been developed as a separate system is that many of its features are implemented in a "non-standard" way. This may be of concern to some, but the purpose of MyForth is explore new territory with the objective of improving the performance of the 8051 code it generates while also greatly simplifying the Forth development environment.

Charley has been greatly influenced by Chuck Moore and, many of the ideas in MyForth are based on Color Forth. MyForth is a very small and simple Forth implementation, also reflecting Chuck Moore's philosophy.

The result is a Forth system in a very small package that has high performance and all of the tools you need for professional 8051 development.

Viewpoint

This manual instructs the new user in the structure, use and practical application of MyForth. The manual also provides some insight into the rationale and methodology behind various implementation features.

To use this manual effectively, you should be familiar with Forth and the 8051 instruction set. The manual is written from the viewpoint of someone who wants to use MyForth to develop applications, but needs to know how to get started. It also provides reference information for experienced users and for those who want to understand more about how the system works.

This manual was written by a new user of MyForth with the help and encouragement of Charley Shattuck. Although Charley has reviewed the text for accuracy, the organization and content are entirely those of the author, Bob Nash.

The reader is hereby cautioned: this manual is written primarily to meet the author’s need to understand and use MyForth and may not meet the needs of a broad audience. It is not intended as a commercial product.

Although the author is enthusiastic about the capabilities of this system, his viewpoint is independent enough to caution users about unusual and non-standard usages that he encountered while learning about MyForth.

Throughout the manual, the user is encouraged to try coding examples and “learn by doing” -- this soon reveals the power and simplicity of the tools.

Although Charley uses MyForth in a Linux environment, the author works in Windows. Thus, this manual was prepared primarily for readers working in a Windows environment.

Development Environment

Although MyForth can be used with many different 8051 compatible processors, this manual describes its use with Silicon Laboratories processors.

A convenient platform for starting out with MyForth is one of the Silicon Laboratories development systems. These all provide the EC2 serial adapter and software needed to load the MyForth bootloader.

After the bootloader is installed, the EC2 and its software are no longer needed: all MyForth programming is thereafter performed via the serial port. If you already have an EC2 (or know a friend that has one), you can simply buy a Silicon Laboratories Target Board and use it.

Of course, the AM Research Gadget Development System is also a good choice for MyForth development, especially because the Target Boards are less expensive and are small enough to deploy in limited-space embedded projects. Another advantage is that MyForth is compatible with the AM Research Boot Loader.

Scope

It is assumed that the user is already familiar with the 8051 instruction set, assembly language programming and the basics of the Forth. With this background, this manual provides a guide to the coding and interactive testing of both assembly language and Forth routines.

The manual provides:

1. An overview of the initial installation process that primarily is intended for Windows users
2. The basic use of the development tools
3. An overview of the system architecture
4. A description of basic coding techniques, including Forth, assembler, macros, interrupts and in-line assembly
5. Reference material, such as command summaries

The following are outside the scope of this manual:

1. Descriptions of the Forth computer language or the GForth implementation used to host MyForth
2. Operations that apply primarily to a Linux environment

Conventions

File names, directories, command sequences and Forth Words generally appear in boldface type instead of being enclosed within quotation marks. The intent is to make it easier to identify useful file and command information in the text. Where the intent is simply to refer to sequences within a code example, the sequences are enclosed in parentheses.

Because directories, file names, command sequences and Forth Words are already emphasized in boldface type, they generally appear in lower case.

The terms “directory” and “folder” are used interchangeably.

The terms “directory”, “folder”, “file”, “path”, “Word”, “command”, “character”, “byte” and “number” are omitted when the context is clear. Command parameters are denoted by the “< ... >” sequence, similar to that used in Unix documentation.

Terminology

Host and Target

Throughout this manual, the term “Target” denote the Target processor. For example, the Target processor for a C8051F120-TB Target Board from Silicon Laboratories would be the Silicon Laboratories C8051F120 chip or other chips in its family for which the target board is intended: the Target is the chip that will ultimately run your application.

The Host is the PC “Hosting” MyForth and consists of the PC hardware and various programs and facilities working together to provide a development environment, especially GForth. Most references to the Host can be assumed to be references to GForth.

The Host compiles Target code into a Target image and downloads it to the Target. The Host runs a Target interpreter, providing a standard Forth interpretive command line, complete with an **ok** prompt. The programs and facilities residing on the Host include the MyForth system files, GForth, an editor and the operating system.

Tethering

The Host interactively communicates with the Target via a simple mechanism called a “tether.” This term is used because the Target is connected to the Host facilities by a serial communications link and a simple tethering protocol.

The Target contains a minimal amount of code overhead; it remains closely tied to the Host, which provides most of the user interaction via the Target interpreter.

To maintain the tether, the Target runs a simple program, consisting of a simple program that can execute a single command and provide feedback to the Host. The tethering program running on the Target primarily allows the Host to command the Target to jump to a specific address and execute the instructions there. The Target interpreter manages the interaction between the Target and user.

Because the Target only knows how to execute code located at addresses specified by the Host, the Host must maintain the Target’s context, such as the names of the Words to be executed on the Target and their addresses.

Tethering provides convenient user interaction while minimizing Target code overhead. Tethering is explained in more detail in a later section.

Note that MyForth can also be configured to produce a standalone Forth system that resides on the Target.

Words

For those somewhat unfamiliar with Forth, the word “Word” is used in the conventional Forth sense to designate what in other languages is called a subroutine, procedure or function.

Because Forth Words are preceded by a colon, they are also called “Colon” definitions.

Forth Words execute when they are entered at a Forth interpretive command prompt. Although Words can be executed interactively via an interpreter, they can also execute independent of an interpreter. In most applications, you will test Words interactively with the Target tethered to the Host. Later, when your application is ready to be deployed, you can define a startup Word that will run your application automatically when the chip is powered up or reset.

Code Words

Code Words function the same as Words and can be executed from the Forth command prompt. But, unlike Colon definitions, Code Words are coded entirely with assembly language mnemonics or macros.

In more conventional Forth systems, Code Words are often defined by preceding definitions with the Word “code.” MyForth handles this differently. Code Words or in-line assembly definitions are determined by changing vocabulary search orders. This is explained in later sections.

Macros

Macros are assembly language sequences that compile instructions in the Target processor’s code image when they are executed. MyForth handles macros by changing search orders. Macros are defined using the **:m <name> … m;** sequence, as explained in later sections.

The important thing to know about macros is that they just assemble instruction sequences that cannot be executed standalone and must be included inside a MyForth Colon definition to be executed.

In Line Assembly

MyForth also handles in-line assembly language sequences by changing vocabulary search orders with special Words to encapsulate assembler instructions. This process and the special Words are explained later.

2

Installation

Overview

To use MyForth you must first install GForth. It is commonly available on the Internet and installation is straightforward. MyForth assumes that Gforth is installed to the default directories.

You should also install the Vim editor, also widely available on the Internet, as described below in the Editor section.

Install MyForth by unzipping the distribution file in the root directory. This will make a directory named MyForth with several subdirectories. You are then ready to go.

The installation of GForth and Vim should be smooth as long as you install to the default directories. When you are finished installing on a Windows machine, GForth and Vim will both be installed in the normal C:\Program Files directory.

After installing MyForth, it will be in the C:\MyForth directory.

You can get a copy of MyForth at home.surewest.net/cshattuck .

If you are reading this manual, you probably have a copy of MyForth because it is presently distributed only with the MyForth system files.

Windows Development

Because MyForth was developed in a Linux environment, it is designed to execute from a command line. Thus, Windows compatibility is provided with the Windows “Command Prompt.” This is probably not an environment most Windows programmers prefer or are familiar with.

We encourage you to give command line development a chance: there are only a few commands that must be entered at the Command Prompt and the typical development session will mostly occur within an editor or at the Forth interpreter's command prompt.

Operating in a simple environment using a few commands makes development easier and more productive than the more traditional use of a GUI and a custom Integrated Development Environment (IDE). For one thing, your fingers will mostly stay at the keyboard and your focus will be the task at hand.

Windows development can be reasonably convenient with a few simple changes. This section describes these changes and provides an overview of the Windows development process.

Before leaving this topic, we should mention that the default black background of the Command Window can be changed. We highly recommend changing it to white. Because MyForth uses colors to improve readability, they look better on a white background.

To change the background, first right click on the Command Prompt icon (a black window with “C:\” on it). Next, select the “Properties” option. When the Properties dialog opens, select the Colors tab and change the background to white and the text to black.

Application Development

As mentioned earlier, application development is performed from a Windows Command Prompt.

Development commands consist of a small number of batch files, contained in the local development directory, that are executed to compile, download, test and disassemble your code.

For Windows users, these commands invoke Gforth with command line options to load MyForth files. For Linux users, the same functionality is performed by command files with the same name as the Windows batch files, but without ".bat" extensions. The system is extremely simple and there are only four commands: **c**, **d**, **run** and **sees**.

MyForth uses the command line approach because the added complexity of a GUI and a custom IDE is entirely unnecessary for such a simple and direct development environment. Using a command line also makes development in Windows and in Linux almost identical.

We think you will find that a command driven system is much simpler to both use and understand than a custom Windows application. It is also much easier to support under both Windows and Linux.

Editing

Vim

MyForth is best used with the Vim editor. Vim is a free Vi compatible editor that allows you to execute operations from a command line. Besides being free, Vim also works the same in Windows and Linux.

MyForth can be used with other editors. But, if you use another editor, some of the editing operations described in this manual will be unavailable or accessed differently: you must handle these differences yourself. Also, syntax coloring, an important part of MyForth, is easily added with Vim and may be much more difficult with other editors.

Vim is widely available on the Internet. In Windows, install it in the default directory (**C:\Program Files**) and replace a few configuration files with new ones, as described below. The new configuration files are supplied with MyForth.

Usage

If you have never used Vim or a Vi style editor, do not be intimidated by the prospect of learning “yet another editor.” Although Vim is designed and optimized for those familiar with Vi, it can be used like a conventional Windows editor with highlighted cut and paste operations and mouse navigation.

Vim also has pull-down menus for most functions that are normally performed from the editor's command line in a Vi environment. The new user can function quite well by simply remembering to use the **i** command to insert text and to use the escape key to exit the text insertion mode; this takes a little getting used to, but the adjustment is not particularly difficult.

Vim has a number of features that facilitate using MyForth. One of these is the ability to easily navigate between files by placing the cursor on the file name and executing the **gf** (go file) command.

Another indispensable feature is the ability to view and edit the source code for your definitions by putting the cursor on them and pressing **Ctrl-]**.

Tags

An important feature of Vim when used with MyForth is that it can use the **tags** file that GForth automatically updates whenever a Word is defined. Using this feature, you can switch to the file defining a Word of interest as described above.

To use **tags** with Vim, you do not need to change any settings. However, if you do need to change the tag file reference, just set **tags=./<name>** in the **_gvimrc** file. A modified version of this file is furnished with MyForth in the **\MyForth\vim** directory. Put this file in the Vim installation directory (e.g., **C:\Program Files\Vim\Vim64\**).

Navigation

Vim allows you to navigate easily around a set of files and the definitions in them. We suggest starting your editing sessions with **gvim job.fs** even if you know the file you want to edit. From the JOB file, you can easily navigate to just about any file in your application by putting the GVim cursor on the name of the file and pressing **gf** (go file). If you are in a file and want to go to the file that defines a particular Word, put the cursor on the Word and press **Ctrl-]**. If you have navigated to a particular location and want to back out, press **Ctrl-6**.

Another useful Vim operation that uses the **tags** file is the ability to edit a particular definition without having to know where it is defined. To do this, execute: **gvim –t <word>** , where **<word>** is the name of the definition you want to edit. For example, to edit the definition for emit, execute: **gvim –t emit**.

Vim will search the **tags** file for the name of the file containing the definition and open the file with the cursor just below the Word you specified. Of course you must have compiled your application earlier so that the Word appears in the tags list.

Colors

To use MyForth's color highlighting conventions (highly recommended), move the custom version of **forth.vim** from the **MyForth\vim** directory to the **syntax** directory in the Vim installation directory. The tag and color features can be used with other editors, but that is not covered in this manual.

The use of color highlighting greatly improves the readability of MyForth source code and can be considered a "poor man's" version of Color Forth's use of colors.

If you want to use the MyForth syntax coloring with other Forth systems, such as SwiftForth, you may have to add a file that tells Vim about the extension used for Forth source code files. In the case of SwiftForth, the source extension is ".f" and it can be specified by copying **SwiftForth.vim** in MyForth's vim directory to the **C:\Program Files\vim\vimfiles\ftdetect** directory.

3

Projects

Overview

The following describes how to start and develop a new MyForth project. The process is mostly manual, but there is very little to it. Essentially, you will be establishing a project directory, copying files to it and editing a configuration file.

Initialization

Project Directory

MyForth projects are contained in project directories. These directories are created under **myforth** in the **projects** directory. For example:

C:\myforth\projects\myproject

Here are the steps needed to create a new project:

1. Create a new project directory under the **\MyForth\projects** directory and copy all of the files contained in the MyForth directory to it. ***This will give you all of the MyForth system files, but it does not customize your project for use with a particular processor.***

2. To complete your project template, you must select a processor. To do this, copy the files from one of the processor template files to you project directory. For example, if you want to create a project for the Silicon Laboratories C8051F300 chip, copy the files from the **300** template directory to your project directory:

   copy c:\myforth\300\* c:\myforth\projects\myproject

A command sequence such as the one given in Step 2 above will copy a few processor-specific configuration files to your project directory from the **300** directory. These include **config.fs**, **job.fs**, **load.fs** and, possibly, a special function register definition file such as **SFR300.fs**.

Another project configuration option is to copy your files from an existing project with a configuration that is similar to the one you will be using for your current project. This is a bit more straightforward than copying from two separate directories, but does not guarantee that you will be using the distribution versions of the system and configuration files.

Caution is advised when copying files from a previous project directory because they may not be current.

After copying files from the appropriate directories, you can develop your project using just the files in your project directory: development will proceed independent of the system and processor configuration directories.

This ensures that when changes are made to the system and configuration files, your old code will still work. Because every project directory contains all of the files needed to generate and download a working application, your application will always work, regardless of any later changes to MyForth.

Note: because of the small size of a complete development image (about 200K), you can copy both Linux and Windows files to your new project directory. This also allows you to easily change environments later.

Examples Directory

To make it easier for you to compile and disassemble examples given in this manual, MyForth is distributed with an **examples** project directory (e.g., **\myforth\projects\examples**). In this directory, various example definitions are contained in **examples.fs**.

The **examples** project directory is configured for a SL C8051F300 Target and illustrates a project directory that has been configured for a specific processor as described above.

If you are connected to a 300 Target such as an AMR 300 Gadget processor, then you can also interactively test the examples in this manual. Because the **examples.fs** file does not contain any processor-specific code, it can be used with other processors by creating and configuring a project directory, as described above, and copying **examples.fs** to it.

To make it easier to change to the **examples** directory, the MyForth system files include a batch file, **examples.bat**, that changes to this directory. You can put this in your root directory or, if the **myforth** directory is in the command path, you can execute it from any command prompt window. You may want to provide other similar batch files for your projects to allow you to quickly navigate from project to project.

Configuration

Overview

Before starting to develop your project you should examine and perhaps change the configuration of your system by editing the **config.fs** file (Appendix A provides a typical listing).

To make changing the configuration file easier, it is useful to understand the location and operation of the Boot Loader, interrupt vectors and your application code. The Boot Loader chapter discusses these topics.

However, if you copy the configuration files from the appropriate processor configuration directory, such as **300** or **120**, your configuration should be close to being correct. Mostly, you will have to decide what kind of interpreter you want to use and edit the configuration file, **config.fs**, accordingly.

Interpreter

MyForth allows you to interact with the Target processor via a tether or with a standalone interpreter that resides on the Target. With a Target-resident interpreter, the names of executable Words and their execution vectors are stored on the Target to form a simple dictionary.

When headers are compiled on the Target, the Target image is larger, but the Target can operate standalone with its own interpreter. This allows you to exercise Target functions with a dumb terminal without a connection to a Host PC running MyForth (i.e., without a tether).

Normally, you will develop using a tether. In this case, the first line of **config.fs** that reads: **true constant tethered** should be left alone. If you want to compile heads on the Target and install a Target interpreter, change “true” to “false.”

Reset Vector

MyForth's Boot Loader resides at the processor's normal reset vector at $0000. It normally checks for download requests and, if there are none, it times out and jumps to the tethering code or to your application.

Code that executes after bootup can reside at several different locations, depending on the Target processor. For most Silicon Laboratories chips, the reset vector will be location $200. Because of the larger page size for the C8051F12x series processors, the location is $400. Other targets may have different reset vectors.

ROM Start

You may also want to specify the start of ROM, the location where MyForth and your application program will start. As provided, only the first three bytes of the start of ROM are used for interrupts: application code starts after the Cold Start vector.

If you will have interrupts vectors other than the Cold Start vector, you must change the ROM start location to allow for them. This re-allocation could be automated, but the price would be added complexity.

The MyForth rationale for manually changing the start of your code is that the application programmer will certainly be aware of the addition of an interrupt vector. Consequently, the requirement to change system files to match is not particularly onerous.

Not reserving a block of memory for all possible interrupt vectors also saves some memory. Most programmers wouldn't bother, but MyForth programmers have the choice to either waste memory or tighten their code.

Target Size

The target size specification is normally set correctly for the amount of Flash available in your processor. This is used when allocating the Target memory image (at **target-image**) and in writing the Target image files (**chip.bin** and **chip.hex**). For example, if your processor has 64K of memory, $10000 is the correct target size.

MyForth does not check for compilation past the end of ROM. It does display the total ROM used at the end of each compilation and the Host stack. You can use these indicators to determine if there is a compilation error needing your attention, such as exceeding available ROM.

Baud Rate

The baud rate for MyForth is set in **serial-windows.fs** or **serial-linux.fs**. Normally, the baud rate is set to 9600 for compatibility with the AM Research boot loader.

However, when using the C8051F120 chip, the Boot Loader is a custom MyForth Boot Loader. It sets the chip to run at 98 MHz (4 X 24.5 MHz) and the serial rate is set to 38.4K baud. If you have copied the configuration files from the **120** configuration directory, the baud rate should be set correctly.

Loader and Job Files

When you compile an application with the **c** or **d** commands, they run GForth and include **loader.fs**. This file sets up terminal color options and includes all of the files needed to build MyForth. The following is a list of the files included by the Loader file with some comments telling what they do:

```
include vtags.fs use-tags        \ set up tags file for GVim
include config.fs                \ configure the application
include compiler.fs              \ load the MyForth compiler
include saver.fs                 \ code for chip.bin and chip.hex write
include dis5x.fs                 \ disassembler
include download-cygnal.fs       \ downloader definitions
include dumb.fs                  \ command line dumb terminal

\ Forth primitives.
include misc8051.fs
rom-start org

\ This is the application.
include job.fs

report                           \ report compile results
save                             \ save chip.bin and chip.hex
[ .( Host stack= ) .s cr         \ display the Host’s stack
```

These files will be discussed more in later sections. For now we want to focus on the **job.fs** file (Job file). It is loaded as a convenient way to modularly load your application.

Typically, to build a complex application, you will include several files, using an **include <filename.fs>** command sequence in the Job file. This will load your application’s source code files. Thus, the Job file will include library files, device drivers, utilities and application source to build your application from functional modules.

Using the Job file to include major functional components is a convenient way to load your application in the proper sequence and to organize it. We suggest that you start your editing sessions by executing **gvim job.fs**. By doing this, you can easily navigate to your application’s source code files by placing the cursor on the source code file name and typing **gf** (go file). You can nest this command as deeply as you wish to access other files defined in the target file. You can back out by executing the “Ctl-6” key combination when you compile your application. For small projects, you can include all of your source code in the Job file.

Here is the content of a basic Job file:

```
\ job
\ this is a comment line
include sfr300.fs
include io.fs
include myproject-utilities.fs
include myproject.fs
: go   init   begin  doit  again
```

Note that the first and second lines are comments. Comments are formed by a backslash character followed by a blank character.

The next two lines load the Special Function Registers (SFRs) that are required for a particular processor. In this case, SFRs for the Silicon Laboratories C8051F300 chip are loaded. The **io.fs** file is optional, but typically contains definitions that map processor pins to peripherals or otherwise configure the I/O for an application.

The two lines that follow the I/O configuration will load other source code files that make up most of the application.

The last line illustrates a definition appearing in the Job file. In this case, one could assume that the definitions for **init** and **doit** are contained in either **myproject-utilities.fs** or **myproject.fs**.

The last definition in the Job file is **go**. For applications that will automatically start up (i.e., turnkeyed applications) this is the *Startup Word*. The naming of the Startup Word is just a MyForth convention. You can change it by editing the definitions that establish the interrupt definition for turnkeying the Job file:

```
\ --- Finally patch the reset vector --- /

\ Turnkey or interactive.
\ start interrupt : cold stacks init-serial go ;
start interrupt : cold stacks init-serial quit ;
```

The above code illustrates another function of the Job file: it contains options that you may want to change for your application. These are best done where there is more visibility to the programmer than in the configuration file.

Usually, the definition of **go** will include application startup initialization (e.g., **init**) followed by the main application Word, such as **doit**, that executes an endless processing loop (the **begin … again** loop).

Some programmers prefer to put the **go** definition in **myproject.fs** where all of the Words in its definition are defined.

The main application file in this example is named **myproject.fs** to make it clear that this file contains the main application. In many MyForth projects this file is named **main.fs**, but this is a convention, not a requirement. Appendix A lists a Job file from an actual application.

Please note that the Job file is one of the configuration files loaded from a configuration directory (e.g., **120**). Primarily, the **job.fs** file is processor specific because of the inclusion of the SFR definition file. For some chips such as the AduC816, the serial port code must also be changed.

Note: although the Job template file includes the SFR and serial definitions that are appropriate for a selected processor, it must be edited to include your application files.

Compiling

To compile an application, first ensure that all of your source code is contained in the Job file or in application files that are “included” in the Job file. To compile the Job file, execute the **c** command from the PC’s Command Prompt.

The following shows the results for a typical compile from the command window.

```
C:\MyForth\test>c
C:\MyForth\test>”\Program Files\gforth\gforth.exe” serial-windows.fs
loader.fs –e ‘open-comm target talking’
HERE=2180
Host stack= <0>
Talk to the target
```

The most important information, other than possible error messages, is the amount of flash program memory that will be used by your application. For this example, the application ends at hex location $2180 (HERE refers to the Target).

The location of HERE on the Target is measured from address 0 and includes the boot loader, interrupt vector tables, debug utilities (if loaded) and your application code.

Note that the first lines echo the contents of the **c.bat** file. You can see that it invokes Gforth, loading the appropriate serial communications file and **loader.fs**. The sequence beginning with **–e** tells GForth to execute the following quoted command sequence. The commands in the sequence open the serial port and start up target communications.

You will know that the interpreter is active when you see the traditional Forth ***ok*** prompt when you enter a carriage return.

After downloading you can verify communication with the Target by entering **.s** (print stack) to request the Target to display its stack contents (e.g., <0>). Note that responses from the Target will appear in red.

You may also want to execute **words** to display the words that were just compiled. These commands can be entered at the command line for execution by the Target.

After downloading your application, you can immediately start testing it. You can also test your application immediately after you power up a Target with a downloaded project in it: your project code is stored in Flash along with the tethering code needed to talk to the Host PC.

Thus, you may just want to use the **c** command to establish the tether and test existing code in the Target. If you want to use **c** to recompile the application and disassemble some code, the Target does not have to be active.

The tethering routine is active whenever the Target is active. If your program “hangs up” while executing some errant code, you can press the reset button or cycle power to the Target board to restore control at the Target interpreter. In some cases, you may have to kill and restart the Command Prompt window.

Remember that the interpreter is talking to your application via a simple tether routine executing on the Target.

If you edit the Job file so that the compiled program executes the your turnkey Word, your processor will automatically start up executing the turnkey Word (e.g., go). Generally, you will not be able to interactively test with a turnkeyed program because it is executing your project code within an infinite loop and it will not respond to the tethered interpreter. However, you can interact with a turnkeyed program over the serial port if you configure your application to run standalone with an interpreter and dictionary resident on the Target. A later section explains this process in more detail.

Here are some compiling facts:

1. The **c.bat** file calls **gforth.exe**, which includes **loader.fs**. This file (see Appendix A) loads a number of system files, such as the compiler and disassembler, and then loads **job.fs**, which includes SFR definitions, your I/O configuration and your application.

2. The compiler always produces two auxiliary files, **chip.bin** and **chip.hex**. The **chip.bin** file is a binary load image and **chip.hex** is an Intel Hex representation of the image that is suitable for use with a programmer (e.g., the EC2 from Silicon Laboratories). Note that Vim can display files in hex so you can examine **chip.bin** with it. The **chip.hex** file is text file in Intel hex format and can be examined directly with Vim.

3. The 2180 bytes used in the example includes the entire MyForth system residing on the Target. Normally, applications will grow very slowly past this point because many of the Forth routines in the target image can be re-used by calling them from your application.

Decompiling

see

After compiling, you can view the assembly code for a definition by entering **see <word>**, where **<word>** is the name of the definition in your application. To test this, try disassembling one of the words listed in the **words** dump.

For example, try decompiling the definition for **emit** by entering **see emit** . The following shows the output of the **see** decompiler:

```
---------- emit
        0403  30 99 FD     jnb SCON.1,0403 emit  if.
        0406  C2 99        clr SCON.1
        0408  F5 99        mov SBUF,A  #!
        040A  E6           mov A,@R0  (drop
        040B  08           inc R0  drop)
        040C  22           ret  ;
```

The disassembly of the definition is displayed one line at a time, as you press any key but “q” or escape. It is generally convenient to press the space bar or enter **n** (next) to advance the decompiler display. The display will continue until you reach the end of the processor memory (whew!) or until you enter **q** (quit) or press the escape (Esc) key. You can also (less gracefully) exit the decompiler by entering **Ctrl-c**.

Observe that the definition starts at Hex location $0403 and ends at $040C. Most of the definition consists of macros, many of which are designed to perform operations needed to build MyForth. But, note the absence of calls. This is because macros are executed when named within a definition to lay down code; thus they appear in your Target code when needed: they are not defined elsewhere in the Target image and then called from within the definition like a subroutine.

This is one significant difference between MyForth’s approach and that of more conventional Forth systems. Of course you can define routines and call them. If a routine is used often and the overhead of a call does not reduce your application’s performance, this may be desirable. However, with increasing amounts of flash memory available in modern 8051 processors, this is less important than in the past.

Also note how little memory is consumed in this definition (10 bytes).

Last, please note the use of color to visually aid the interpretation of the disassembly. The name of the disassembled word is listed in red after some dashes. This helps identify the location of entry points.

Addresses are listed in black and the compiled bytes are listed in blue. The actual decompiler output is in green, followed by the compiler's attempt to identify the name of the macro that produced the code (in black).

decode

To decompile starting at a specific address, use **<address> decode**, where **<address>** is the address of the start of the disassembly. The default mode is Decimal; to specify a Hex address, prefix the address with "$."

For example, assume you want to decompile your application starting at Hex location $400. Entering **$400 decode** would display the results given in Figure 2 above, but with the following additional line at the beginning:

```
0400   02 08 7B      ljmp 087B  cold  ;
```

Now you can see that emit is the first definition after the Cold Start vector. In this example, the processor is a C8051F120 and its startup code is at $400 because of its larger flash page size.

This example was chosen to start at a known entry point. The decompiler is somewhat smart, aligning decode operations at sensible start points, but entering an arbitrary address or starting at an address containing data may yield raw code without reference to named code entry points.

As with the **see** command, each line appears as you enter keys such as the space bar or **n** (next). Terminate the decode display by entering **q** (quit), escape (**Esc**) or **Ctrl-c**.

Downloading

To download your compiled application to the Target, connect your PC's serial port to the Target development board (e.g., a Silicon Labs Target Board) and enter the **d** command from the PC’s Command Prompt.

If this does not work, check that the Target board is plugged in and that the PC’s serial baud rate is set to the same as that of your Target. The standard baud rate for MyForth is 9600, except for a C8051F120 Target board running at 98 MHz. In this case, the baud rate is four times normal (38.4k baud).

As the **d** command executes and the download proceeds, you will be prompted for actions at each stage (e.g., press the reset button). As your project code downloads, the downloader will display the number of pages of flash memory that have been downloaded to the Target.

When the download has finished, you will be talking to the Target.

Note that the compilation and interactions of ***c*** *command and the* ***d*** *command are identical, except that the* ***d*** *command first downloads your compiled application before starting to talk to the Target.*

Tethered Operation

Passing Parameters

You can pass parameters to the program residing in the Target by putting numbers on the Target's stack. To do this, just enter the numbers on the Target interpreter's command line, followed by a # sign. The Word # is a GForth Word that executes to compile code in the Target's image that will put a byte on the Target's stack when it is executed. to put a byte on its stack.

To enter a 16-bit value (e.g., an address), follow the number with ## (also a Word). Using these Words following a number is a bit different from most Forth systems that put numbers on the stack without a # or ##.

The reason for using these Words after numbers is that it greatly simplifies the compiler and tethered interpreter.

Stack Display

As with most Forth systems, you can display the stack contents with **.s**. It is often a useful check to see if the Target is responding by entering **.s** at MyForth's ***ok*** prompt.

Defined Words

As mentioned earlier, you can display the Words defined on the Target by entering **words** at the ***ok*** prompt.

Exiting Forth

To exit the Target interpreter invoked by the **c** or **d** commands, simply type **bye** at the ***ok*** prompt.

Note that, if your application appears to "hang up", you are probably no longer communicating with the Target (reset it or cycle power to it). However, you may be able to execute some commands from the Target interpreter's command line, including **bye**.

In some cases, you must restart the Command Window (e.g., if the serial port hangs up). This is an unfortunate side effect of running the serial port in the Command Window. This is normally not required when running under Linux.

Turnkeying

To turnkey a compiled and tested application, edit the **job.fs** file so that the application starts up executing the turnkey Word (normally named **go**). Here is the pertinent code:

```
\ --- Finally patch the reset vector --- /

\ Turnkey or interactive.
start interrupt : cold stacks init-serial go ;
\ start interrupt : cold stacks init-serial quit ;
```

The Word to start and run your turnkeyed application, typically named **go**, is usually defined near the end of your Job file or as the last definition in your main application file (e.g., **main.fs**).

Dumps

You can request the Target to send you a line at a time dump by putting a Target address (a double number) on the Target's stack and executing the **d** command from a MyForth command prompt. The **d** command works in much the same way as **see**, outputting a line whenever an "n" key (or any key but Esc) is pressed. The dump is terminated by pressing "Esc" or "Ctrl-c."

The next section shows how you can use a script file to perform a dump from the Windows Command Prompt and save it to a file.

Scripts

You can execute commands that you would normally enter on the MyForth command line by including them in a text file and executing them with the **run** command from a Windows Command Prompt (see **run.bat**).

Here is an example taken from **script.fs** in the MyForth distribution:

```
\ script.fs
\ An example script which dumps the first 256 bytes of memory.
\ Use redirection to capture in a file.
0 ## d
[ : lines ] 0 do  cr n  loop [ ; ]
15 lines cr
```

This example may not be clear just yet – you may have to read ahead to understand the usage of the left and right bracket, the function of ##, etc.

The line containing "0 ## d" puts a double (16 bit) number (0) on the Target's stack and performs a dump with the **d** command. In response to the **d** command, the Target outputs one dump line and waits for another command. The code arranges for the next command to be a "cr." After this, it executes an "n" to progress to the next line.

Because this code is being executed in a script file, there is no user to press "n" to request more than one line. Thus, **lines** is defined (in GForth) to execute "cr n" 15 times.

You may wonder why the sequence “0 do cr n loop” appears just after the right bracket. As you will soon be shown, the right bracket establishes the Target vocabulary: Words following the right bracket are searched for in the Target vocabulary first, followed by the Forth vocabulary.

As you will soon find out, **do … loop** is not a MyForth looping construct so it may seem a bit strange to have it follow a right bracket. In this example, most of the “0 do cr n loop” will be defined on and executed by the Host, GForth. This is because most of the Words such as **do** and **loop** will not be found in the Target vocabulary: they will be found and compiled when they “fall through” to the Forth vocabulary.

However, **n** and **cr** are defined as Target Words: they send characters to the Target to signal “send a carriage return character” and “deliver the next line of a dump.” When **lines** executes, GForth loops, requesting the Target execute “n.” Note that the “cr” in “15 lines cr” is executed on the Host, but has the same effect as executing on the Target (if you can wrap your mind around that).

Normally, of course, you would execute **d** from a MyForth command prompt and not a script. The above shows how a dump can be performed from a Command Prompt window so that its output can be redirected to a file for printing or documentation (e.g., **run script.fs >mydump.txt**).

4

Compiler

Overview

This chapter describes how to develop a program using a few simple commands entered from a Command Prompt. Normally, you will develop your program and compile it with the **c** command. If you want to compile and download your program to the Target, you can use the **d** command.

Although there are a few other utility commands that you can execute from the Command Prompt, you will mostly be using either **c** or **d**.

The following sections also describe the usage, implementation and mapping of processor resources. Topics covered include:

1. Memory and stack mapping
2. Forth implementation
3. MyForth programming

Although assembly language statements can easily be incorporated into your Forth or macro definitions, this topic is covered in the Assembler chapter.

Mapping

Memory

MyForth uses processor RAM for registers, variables and the Forth data and return stacks. The following sections describe where these are located and how they are used.

The memory map for each processor is a bit different, depending on the amount of RAM available. All of the Silicon Laboratories processors have at least 256 bytes of RAM, but MyForth supports processors with as little as 128 bytes of RAM.

Boot Loader

The MyForth Boot Loader is located at location $0000. Because the Boot Loader occupies memory that is normally used by the processor interrupt vectors, the Boot Loader re-maps the interrupt vectors to start at $200 (or $400 for the C8051F120). The Boot Loader is explained in more detail in a later chapter.

Programs

Your programs are stored starting just after any re-mapped interrupt vectors. This will be just past location 512 in most Silicon Laboratories chips or just past location 1024 on the C8051F120.

You can find the exact location of your program code using the **see**, **sees**, **decode** or **d** (dump) commands on one of your definitions. Note that your programs will start just after MyForth system definitions (e.g., **emit**).

Stacks

The location of the data and return stacks varies, depending on the number of direct cells available.

For chips with just 128 bytes of RAM, the return stack starts at $21, to leave room for variables and bit variables, and grows upward (increasing addresses) toward the data stack. The data stack for these chips starts at $80 and grows downward toward the return stack.

For chips with 256 bytes of RAM, the return stack starts at $7f and grows upward toward the data stack. For these chips, the data stack starts $fe and grows downward toward the return stack.

Implementation

Threading

MyForth is a subroutine (call) threaded Forth. Note that many named sequences are defined as macros and used very much like you would normally use Forth Words. Macros are compiled directly into the Target image without a call and thus cannot be executed standalone like a Word.

When a call is immediately followed by a return (**ret** instruction), the call is changed to a jump and the return is not compiled: the called routine will perform the return. This optimization saves memory and increases speed. This is efficient tail recursion, but also works as a goto.

Vocabularies

Don’t skip over this section.

MyForth has only two vocabularies, Forth and Target. When Forth is searched, Words found are executed by Gforth; when Target is searched, the MyForth Target compiler executes the Words (or macros).

Vocabulary search order is controlled by two Words, **[** and **]** . The secret to MyForth’s simplicity and power is largely due to the judicious use of these two Words.

For Forth aficionados, here are their definitions, taken from the file **compiler.fs** listed in Appendix A:

```
: ]  only forth also target also definitions ; immediate
: [  only target also forth also definitions ; immediate
```

From these you can see that **]** establishes the Target vocabulary first in the search order, followed by Forth. Of course, **[** does just the opposite: Forth is searched first, followed by Target.

If you examine the source for MyForth, you will see these two Words used in a variety of ways to flexibly reference either the GForth compiler or the MyForth Target compiler. These can be invoked both inside and outside definitions to control what is compiled or executed and to select the compiler that performs the operations. If you understand these two Words, you will understand most MyForth definitions.

In MyForth source code you will mostly encounter these two vocabulary switching Words and the “:” and “:m” defining Words. Almost everything else is defined using thes four Words. As you use MyForth, you will learn how they can be used to efficiently and flexibly control the generation of 8051 code.

Words

Forth definitions (Words) can be coded as you would code them with most other 8-bit Forth implementations. To define a new MyForth Word, use the **: <name>… ;** defining sequence.

The body of Colon Words defined in MyForth consists of a combination of previously defined MyForth Words or macros. With the judicious use of the **[** and **]**, you can also access the assembler and the GForth compiler.

Remember, that Colon Words execute on the Target. Normally, the Target interpreter exercises these Words, but MyForth also allows you to put the name headers on the Target and build a Target that has its own standalone interpreter.

Macros

Often, in-line assembler sequences are not the best way to code for readability or efficiency. If there is a sequence of assembly instructions that performs a specific operation or that is used repeatedly, it is often better to code the sequence as a macro.

Macros, although they have a name, cannot be executed except within the context of another definition.

A macro is just a sequence of instructions that are given a name. When this name appears in a definition, the macro executes immediately to compile instructions or data in the Target image.

The Target image is a block of memory residing on the Host starting at **target-image**. It is an image of the bytes that compiled by the **c** and **d** commands. This image will also be downloaded to the Target when you execute the **d** command. The image is also written to **chip.bin** and **chip.hex** whenever you compile using **c** or **d**.

Similar to the sequence used for defining Colon Words, macros are defined with the **:m <name> … m;** sequence.

We suggest looking at **misc8051.fs** in Appendix A to see examples of how macros are defined and used. Note that a large part of MyForth is built with macros.

Here are some simple macro definitions listed under “Stack Operations” in **misc8051.fs**:

```
:m dup      s dec $f6 , m;
:m swap     $c6 , m;
```

In the definition of **dup**, the code for decrementing **s**, the stack pointer, is laid down in the Target image, followed by a one byte instruction, $f6, that is also put in the Target image with a “**,**” (comma). Decrementing **s** changes the stack pointer to point at the next (added) stack item. The $f6 instruction code decompiles to **mov @R0, A**. This moves the top of stack into the new cell that **s** now points to; this cell is now the second item on the stack. Thus, to duplicate the top of stack (contained in **t**), the stack pointer is decremented and the byte contained in **t** is moved into the cell pointed to by **s**.

You may have observed that the instruction byte for the indirect move was put directly in the Target image without resort to an assembler sequence. This is in keeping with MyForth's theme of simplicity: it is a one-time look-up for the programmer and should not require an assembler. In MyForth, most of the effort is put into the disassembler, which you can use to verify your coding.

Moving to the definition of **swap**, you can see that it too consists of an instruction byte that is laid down in the Target image. The instruction is **xch A, @R0**. This exchanges the contents of **t** (top of stack, the accumulator) with the contents of the cell pointed to by **s** (**R0**, the stack pointer).

Looking at other macro definitions near the definitions of these two macros, you will notice a number of things that you undoubtedly don't understand right now. These will be explained later.

But, before leaving the macro definitions, it may be instructive to examine the definition of **nip** near the definitions for **dup** and **swap**. You can see that code within a macro can contain assembly instructions: not all system macros are built by directly writing bytes into the Target image. The assembler definitions available to you are covered in the Assembler chapter.

You may be wondering how definitions like **dup** can be executed from the interpreter if they are defined as macros. They can't. Typically, when your application is compiled, a file named **interactive.fs** will be loaded after your application is compiled. It contains normal colon definitions for common macros such as **dup**, **swap** and **drop** so that you can execute them from the command line.

If you compile an application containing **interactive.fs**, you will see that definitions like **dup** are in the list produced by **words** and thus can be executed at the Target interpreter's **ok** prompt.

If you decompile one of the definitions for **dup**, **drop** or **swap**, you will see that the definitions contain the exact code given above for the macro versions but the code for each is terminated with a **ret** (return) instruction. This makes the definitions callable routines. If your application is complete and you no longer intend to exercise it from the tethered interpreter, you can comment out the line that loads **interactive.fs**.

Registers

Here are the definitions for registers and Special Function Registers (SFRs) as they are defined in **misc8051.fs** :

```
\ ----- Virtual Machine ----- /
\ Subroutine threaded.
  0 constant S \ R0 = Stack pointer.
  1 constant A \ R1 = Internal address pointer.
$e0 constant T : .T T + ; \ Acc = Top of stack.
\ DPTR = Code memory address pointer, aka P.
\ B is used by um*, u/mod, and over, not preserved.

\ ----- 8051 Registers ----- /
$82 constant DPL $83 constant DPH
$98 constant SCON : .SCON SCON + ;
$99 constant SBUF
$80 constant P0 : .P0 P0 + ;
$90 constant P1 : .P1 P1 + ;
$a0 constant P2 : .P2 P2 + ;
$b0 constant P3 : .P3 P3 + ;
$81 constant SP
$d0 constant PSW : .PSW PSW + ;
$88 constant TCON : .TCON TCON + ;
$89 constant TMOD
$8a constant TL0 $8b constant TL1
$8c constant TH0 $8d constant TH1
$8f constant PCON
$a8 constant IE : .IE IE + ;
$b8 constant IP : .IP IP + ;
$f0 constant B  : .B  B + ;
\ $fd constant SP0 $80 constant RP0
$100 constant SP0 $80 constant RP0
```

Note that definitions starting with a “dot” allow you to specify individual bits within ports and cells.

Here is a summary of MyForth register and pointer usage:

0	**R0** (**s**) 8 bit stack pointer.
1	**R1** (**a**) addressing index register – naming is from Color Forth
2	**R2** Scratch register
3	**R3** Scratch register
4	**R4** Scratch register
5	**R5** Scratch register
6	**R6** Scratch register
7	**R7** Scratch register

Acc (**t**)	Top of stack
b	Can be used as a scratch register, but it is used by **um***, **u/mod** and **over** (it is not preserved)
DPTR (**p**)	Data Pointer – can be used as a scratch register

Assembly definitions or macros must preserve or knowingly and carefully change the virtual machine registers **t** (accumulator) and **s** (stack pointer, **R0**). Generally, the **a** (address) register, R1, is used as an indirect address pointer and does not need to be preserved between definitions. However, you should be aware that it may contain a pointer that want to preserve within your definition.

Note: The naming of ***a*** *was taken from Color Forth – it should not be confused with the Accumulator, which is named* ***t*** *(for top of stack).*

All other registers may be changed freely and need not be restored, but these registers should not contain static data: any register may be used and modified by any other Forth or assembler definition. Static data should be kept in direct cells.

Here is a list of other processor resources that have been defined for use by your definitions:

DPL and DPH
SCON, SBUF, TCON, TMOD, PCON
IE, IP
TH0, TL0 and TH1, TL1
SP, PSW, SP0, RP0

Refer to the listing for **misc8051.fs** in Appendix A for more detail on the definition of the above resources. They are defined in the sections near the top of the file named “\ ----- Virtual Machine ----- /” and “/ ----- 8051 Registers ----- \.”

Data Stack

Implementation

The top of the Forth data stack is held in the accumulator. In MyForth, the designation of the top of stack is **t** (for top). The following sections describe some Data Stack operations.

#, ~# and

If you want to put a number on the Target's data stack at run time, follow the number with **#**. The action of **#** is to use a number from the Host's data stack to create code that will put it on the Target's stack when the definition is executed (i.e., to compile a literal). To put a bit inverted version of your constant on the stack, use **~#**; this is often used to perform logical operations on Special Function Registers, setting and clearing individual bits.

The Word **##** will put a 16 bit number (usually an address) on the Target's stack as two bytes, with the MSB in **t** (i.e., on the top of the stack). Here is an example:

```
: tadr  $0123 ## ;
```

This will put $01 in **t** and $23 under it.

#@, **(#@)**, **#!** and **(#!)**

To fetch data from a direct cell, use #@; to store data to a direct cell, use **#!**. Here are some examples:

```
: set5   $23 #  5 #! ;          \ store $23 in direct cell 5
: get5  ( - n)   5 #@  ;        \ fetch contents of direct cell 5
```

Executing **set5** will store $23 in direct cell 5; **get5** will fetch $23 from direct cell 5 and put it on the Target's stack.

You can use **(#@)** to move data directly into **t** from a direct cell without first doing a **dup**. This is equivalent to performing a move of direct data with the assembler. Similarly, you can use **(#!)** to move data to a direct cell without affecting **t**: it does not perform a **drop** after moving data from **t**.

stacks

To reset both the data and return stacks, include the **stacks** macro in your definition. Note that this is a macro and is not directly executable from the MyForth command line.

Return Stack

Implementation

The return stack is contained in internal RAM and uses the 8051's Register 0 as the stack pointer. This is also named **s**.

push and **pop**

The Target Words **push** and **pop** move values between the data stack and the return stack and may be considered synonymous with the Forth Words **>R** and **R>**, respectively. Although Chuck Moore has used **push** and **pop** for the past 20 years or so, these are not ANSI Forth Words.

Note: **push** and **pop** are also 8051 assembly language instructions that are defined in the assembler. These act on the 8051 processor's stack pointer, **SP**, and do not involve the Target's stack. To choose between these two versions in an application, you can use **[** and **]** to set up the appropriate vocabulary.

For example, here are the definitions for **push** and **pop** for the assembler:

```
[ \ These are 'assembler', not 'target forth'.
: push $c0 ] , , [ ;   : pop $d0 ] , , [ ;
```

And, here are the definitions for **push** and **pop** for the Target:

```
:m push [ t push ] drop m; :m pop ?dup [ t pop ] m;
```

The assembler definitions put the 8051 instruction bytes for **push** and **pop** on the Host's stack and then change to the Target vocabulary to place them in the Target image. These instructions can act on any direct cell, moving it according to the 8051's stack pointer.

*The Target versions of **push** and **pop** act only on **t** (the accumulator).*

Address Register

a and **a!**

MyForth uses Register 1 (R1) as the “address register”, **a**. When **a** is used in a definition, the byte contained in the direct cell address in **a** (R1) is moved to the data stack. Usually, the contents of **a** will be a direct cell address used to indirectly access data.

You can use **a!** to load **a** with a value, as shown in the example below.

@, @+, ! and **!+**

If you have a direct cell address in **a**, you can fetch data from that cell and put it on the data stack using @. Similarly, you can store data from the data stack indirectly to a cell using **!**. Accessing data this way is very useful, especially if you are manipulating data in sequential cells.

To make this process of sequential access even easier, @**+** and **!+** are provided to fetch and store while auto incrementing the contents of **a.** Here is an example of how to store and fetch three sequential bytes, starting at direct cell 5:

```
: put3    5 # a!   $aa #  $bb #  $cc #  !+ !+ !+  ;
: get3  ( -- n1 n2 n3)    5 # a!  @+ @+ @+  ;
```

Data Pointer

|p, **|@p**, and **|@p+**

MyForth uses **p** to designate the 8051 data pointer and provides several macros and Words for managing it. The macros for managing **p** are preceded by a vertical bar to indicate they are “inline” or “macro” definitions. Normally this is how you will use them, but you can make them callable by making them colon definitions (e.g., to save memory if they are referenced several times in your code). Refer to the source code listing in the chapter on the Standalone Target for examples.

The **|p** macro puts the 16-bit contents of the data pointer on the Target’s stack. Often this will be used to save contents of the data pointer prior to changing it so that it can be restored later (e.g., with “|p push push”). This is illustrated in the definition of **interpret** in the code for the Standalone Interpreter in a later chapter. In **interpret**, the contents of **p** are changed in the process of searching the dictionary; when the operation is complete, **p** is restored.

To get data from the location contained in **p**, use **|@p** or **|@p+**. These will both put a data byte on the stack from the location in **p**, but **|@p+** increments the data pointer after the fetch. Examples of how these are used are contained in the definitions of **match**, **find** and **interpret** in the Standalone Interpreter.

p+, **p!** and **##p!**

To increment **p**, use **p+**. To store a new pointer in **p**, use **p!**. ***Caution: both of these definitions are macros: they can be used within definitions but cannot be executed directly from the command line.***

If it is your intent to compile code that will directly set **p** to a particular value, use **##p!**. This is commonly used when you do not intend to manipulate **p** using the Target’s stack, just ensure that the data pointer is set to a particular value. An example of this is contained in the chapter on the Standalone Interpreter in the definition of **dict**.

Variables and Constants

MyForth does not have any dedicated Target Words for defining variables or constants.

There is no "constant" because you can define a Word on the Target that behaves like a constant using existing MyForth components. For example:

```
: five  5 # ;  or :m five  5 #  m;
```

The above definition is not very useful, however, and you would not typically use it in your code when you just want the convenience of using a named constant.

Note that **constant** is available on the Host (GForth) and can be useful in defining Target Words when your intent is just to have the convenience a named constant or variable (direct cell).

Although not all of the Words below have been discussed yet, here is a simple example you can use the Host word **constant** to define some Words. We suggest that you download, disassemble and test these definitions, which are contained in the **examples.fs** source file:

```
\ examples.fs

$0a constant con1
$0b constant con2
5 constant cell5

: test1   cell5 # a!  con1 # ! ;        \ indirectly load cell5 through a
: test2   con2 #  cell5 #! ;            \ directly load cell5 with con2
: .cell5   cell5 #@ h. ;                \ display contents of cell5 in hex
```

In reading the above, it is helpful to know the following:

1. Numbers to be stored in the Target's top of stack, **t**, stack must be followed by # – it compiles the code needed to put the number on the Target's stack when the definition is executed.
2. The Word **!** stores a number that is on the Target's stack into a cell, indirectly through **a**.
3. The Word **#!** stores a byte from **t** into the specified cell address and #@ fetches a byte from the specified cell address and puts it on the Target's stack.

Numbers and Labels

There is no special construct, such as “label”, to define a named memory location in MyForth. However, MyForth allows you to do this if it is needed. You can attach a name to a sequence of bytes, for example, by doing this:

```
cpuHERE constant mycells 8 cpuALLOT
```

In this example, **cpuHERE** returns the current pointer to the next available direct cell. The constant **mycells** is defined on the Host and acts as a label for the start of **mycells**. The “8 cpuALLOT” allots 8 cells after **mycells** by moving the **cpuHERE** pointer. Of course, you must use **mycells** within a Target definition if you want to use it on the Target (e.g., using ##).

If you want to define a Target Word that will put a direct cell address on the Target’s stack, you can simply do this:

```
: cell7  7 # ;
```

It is important to remember that numbers in MyForth put a number on the Host’s data stack; if you want to put a number on the Target’s stack when the definition is executed, you must use #, ##, or ~#.

If you want to label a location, here is an example, taken from **misc8051.fs** that assigns a label to the reset code at location zero:

```
0 org : reset
```

Here is an example of a Target definition that, when executed, will put the address of the current Target compilation address on the stack. This is normally how a “label” is employed:

```
: iamhere   here  [ dup $ff00 and 8 rshift ]  # # ;
```

In the above, **here** puts the Target’s compilation address on the Host’s stack. The left bracket ensures that the following operations occur on the Host: they put the upper and lower address bytes on the Host’s stack in the proper order (MSB on the top of stack). The right bracket ensures that the following operations occur on the Target: the **# #** sequence puts the two bytes on the Target’s stack when **iamhere** executes. Of course you would normally use **##** to put the double number on the Target’s stack in the correct order, but this example illustrates how you can minipulate data on the Host before using it for a Target definition.

Definitons like **iamhere** are seldom necessary. One reason for presenting it here is to illustrate that **here** refers to the Target’s compilation address, not the Host’s. To get the location of **here** on the Host, use **[here]**. The above also illustrates how you can use left and right brackets to change between Host and Target operations.

Interrupts

The most important thing to remember about interrupts in MyForth is that you may have to edit the configuration file, config.fs, when you define a new interrupt. This ensures that the Target compiler will start your application code after the last interrupt in the remapped interrupt area. For most Silicon Laboratories chips, the remapped interrupts will start at $200; for the C8051F12x family of chips, they will start at $400.

To define an interrupt, use **interrupt**. Here is an example and some explanation:

```
start interrupt : cold stacks init-serial quit ;
```

To explain how interrupt works, the components of the above code and the definition of interrupt will be explained individually. Here is the definition of **interrupt**:

```
: interrupt ( a - ) ] here swap org dup call ; org [ ;
```

The **start** before **interrupt** in the example is the address of the start of the remapped interrupt vectors (e.g., $200 or $400). The Word interrupt first saves the current pointer to the Target compiler's image. This address is on put on the Host's stack before **interrupt** is executed, as indicated by the blue stack picture comment in the definition of **interrupt**.

The **]** turns on the Target compiler. The **here** puts the Target image pointer on the Host's stack (its not defined on the Target so it falls through to the Host's Forth vocabulary). The Target image pointer is the location at which any new definition will be compiled.

The "swap org" sets the Target image compilation pointer to **start**.

The **dup** saves a copy of the previous Target image pointer and then compiles a call to it. Thus, a call to the previous compilation address is compiled at **start**.

The copy of the previous compilation pointer is now used to reset the Target compilation address back to where it was before **interrupt** started to execute.

Finally, the cold start definition, **cold**, is compiled into the Target image; thus, the interrupt vector at **start** will point to it.

Conditionals

Overview

The following sections describe the MyForth conditionals. There are not many.

In MyForth, the **if … then** construct uses **t**, as you would normally expect, to conditionally execute code. Like the loop termination conditionals discussed in the next chapter, MyForth provides special conditionals such as **if', if.,** and **–if**.

These do things such as conditionally execute code based on the condition of a bit. The following sections describe each of these, and the code they produce, in more detail.

MyForth does not provide an "else" conditional. By examining some MyForth system code and application examples, you will see that it is not necessary. In fact, you will be hard pressed to find many "if" statements used in the system code or examples. This follows Chuck Moore's practice. It is surprising how seldom "if" is needed.

if and **0=if**

Here is an example of how you might use **if** to conditionally execute code:

```
: iffy1   if  drop  $0a #  ;  then  drop  $0b # ;
```

This will put $0a on the Target's stack if **t** is not zero and a $0b otherwise. Here is how it decompiles:

```
---------- iffy1
06FA  60 03          jz 06FF if
06FC  74 0A          mov A,#0A #
06FE  22             ret ;
06FF  74 0B          mov A,#0B #
0701  22             ret ;
```

This example illustrates a few important points about MyForth. First, you have probably noticed the semicolon in the middle of the definition. In MyForth it is the equivalent of "exit" and just compiles a **ret** instruction. Now perhaps you can see why "else" is not part of MyForth: there are ways to code so that it isn't needed.

Next, you probably wondered why there is a **drop** in front of $0a and $0b. Like the examples with **until** (next chapter), **t** is not automatically dropped. Another way to look at this is that **if** does not consume its argument.

One reason for leaving the top of stack alone is the same as noted for **until**: often you will need to preserve **t**; if not, then the penalty for coding a drop is no more inefficient than always coding it. Along these same lines, including an "else" conditional would make most definitions less efficient by having to include code to jump around the "else" condition.

One other reason for not consuming the argument is that it is very cumbersome to consume it and then make the jump. It can more than double the number of cycles and the code looks very bad when you decompile it. Also, you have removed the cause of the jump, the state of **t**, so you may also need to save that somewhere (e.g., the carry bit) if you need to use it in later processing. It is much simpler and clearer to jump on the state of **t** and clean things up later, if needed.

Enough of that. What about **0=if?** Most of the "if" type conditionals have a counterpart that acts in the opposite sense. In the case of **0=if**, it executes if **t** is zero.

if. and **0=if.**

The **if.** conditional is actually somewhat useful. It executes code based on a bit being set. Here is a reworked version of **iffy1** based on checking a bit.

```
: iffy2  1 .t  if.  drop  $0a #  ;  then  drop  $0b #  ;
```

This will put $0a on the Target's stack if bit one of **t** is set and a $0b otherwise. Here is how it decompiles:

```
---------- iffy2
06B3  30 E1 03     jnb ACC.1,06B9  if.
06B6  74 0A        mov A,#0A  #
06B8  22           ret  ;
06B9  74 0B        mov A,#0B  #
06BB  22           ret  ;
```

The above is a contrived example. Usually you would be checking on something like a bit on an I/O port. Here is an example:

```
: iffy3  1 .P0  if.  $0a #  ;  then  $0b # ;
```

Here is how it decompiles:

```
---------- iffy3
06BC  30 81 05     jnb 80.1,06C4  if.
06BF  18           dec R0 (dup
06C0  F6           mov @R0,A  dup)
06C1  74 0A        mov A,#0A  #
06C3  22           ret  ;
06C4  18           dec R0  (dup
06C5  F6           mov @R0,A  dup)
06C6  74 0B        mov A,#0B  #
06C8  22           ret  ;
```

From the above, you can see that a more efficient (but less clear) definition would be:

```
: iffy3  1 .P0  dup if.  drop  $0a #  ;  then  drop  $0b #  ;
```

If you decompile this, you will see that the **dup** makes room for $0a or $0b on the stack and the **drop**s eliminate the redundant **dup** that # compiles.

The operation of **0=if.** is the same as **if.** except that it executes code based on its bit argument being clear. To see how it works, try defining **iffy3** using **0=if.** instead of **if.** and compare the resulting code with that shown above.

if’ and 0=if’

The **if’** (if carry set) conditional conditionally executes code if the carry bit is set. Here is a simple example that always puts $0a on the stack:

```
: iffy4  $80 #  2*  drop  if’  $0a #  ;  then  $0b #  ;
```

This would decompile as follows:

```
---------- iffy4
06D4  18          dec R0  (dup
06D5  F6          mov @R0,A  dup)
06D6  74 80       mov A,#80  #
06D8  C3          clr C
06D9  33          rlc A  2*
06DA  E6          mov A,@R0  (drop
06DB  08          inc R0  drop)
06DC  50 05       jnc 06E3  if'
06DE  18          dec R0  (dup
06DF  F6          mov @R0,A  dup)
06E0  74 0A       mov A,#0A  #
06E2  22          ret ;
06E3  18          dec R0  (dup
06E4  F6          mov @R0,A  dup)
06E5  74 0B       mov A,#0B  #
06E7  22          ret  ;
```

From the above you can see that it would have been more efficient to use **2*’** instead of **2***. Because the state of carry before multiplying by two (rlc) is not important, the carry does not need to be cleared.

As you would expect by now, **0=if’** (if carry equals zero) conditionally executes code if carry is clear. Here is an example that always puts $0B on the stack:

```
: iffy5  $80 #  2*’  drop  0=if’  $0a #  ;  then  $0b #  ;
```

The code compiled by **iffy5** is similar to that shown above, but a **jc** (jump if carry) instruction is compiled instead of a **jnc** (jump if not carry). As you can see, the conditional jump instruction that is compiled is just the opposite of what you might expect based on the name of the MyForth conditional.

-if and +if

The **–if** and **+if** conditionals execute code based on the state of bit 7 of **t**. In other words, they execute based on **t** being a negative or positive 8-bit integer.

Here is a simple example of how to use **–if** in a definition:

```
: iffy6  ( n – n')  -if  drop  $0a #  ; then  drop  $0b #  ;
```

If you compile and interactively exercise **iffy7**, you will see that putting a positive number such as $7F on the stack and executing **iffy7** will result in a $0b being put on the stack. Try entering the following at the MyForth prompt:

```
$f7 # iffy7 .s
```

This will put $0a on the stack.

The **+if** conditional acts just the opposite of **–f** and will execute code if the value on the stack is positive.

Loops

Overview

The following sections describe how to code loops in MyForth. As you will learn, there are only a few constructs to do this, but they are powerful enough to be all you will need.

Counted

MyForth provides a simple way to implement loops with counts up to 255. The loop counter is held in a cell that must be specified as part of the loop definition. Counted loop definitions start with **<cell> #for** and terminate with **<cell> #next**, where **<cell>** can be a register or direct cell that will be used as a loop counter. The loop count is assumed to be on the stack when the loop starts. The count is put on the stack with **#**, like any other MyForth number.

Of course **<cell>** cannot be **R0**, the data stack pointer (unless preserved). But it can be any other register, direct cell, special function register or even an 8-bit port. The range of the loop is limited to $FF. Here is an example to loop 5 times using Register 7:

```
: init   5 # 7 #for  0 .P2 toggle  7 #next ;
```

This decompiles as follows:

```
---------- init
0691   7F 05         mov R7,#5  #!
0693   B2 A0         cpl A0.0
0695   DF FC         djnz R7,0694  #next
0696   22            ret
```

You can also pre-load **t** with a number and execute this definition of init:

```
: init   7 #for  0 .P2 toggle  7 #next ;
```

It will perform the same function as the first **init**, but it will load **R7** from **t**; this definition is slightly less efficient because the top of stack pointer must be adjusted after **R7** is loaded from **t**.

If you examine the definition for **#for** and **#next**, you will see that **#for** compiles a **mov** to a direct address or a register and **#next** compiles a **djnz** instruction to a direct cell or register, as illustrated in the decompilation above.

Nested

To executes longer loops, you can nest **#for … #next** loops. Here is an example:

```
: delay   0 # 7 #for 0 # 6 #for 50 # 5 #for 5 #next 6 #next 7 #next ;
```

Here is how this decompiles:

```
0700   7F 00          mov R7,#00  #!
0702   7E 00          mov R6,#00  #!
0704   7D 32          mov R5,#20  #!
0706   DD FE          djnz R5,0706  #next
0708   DE FA          djnz R5,0704  #next
070A  DF F6           djnz R5,0702  #next
070C  22              ret  ;
```

Conditional

In MyForth, conditional loops are formed with a **begin … again** type conditional looping construct. Loops can use **t**, as you would expect, to test for loop termination. However, unlike most other Forth implementations, **t** is left untouched when the loop terminates. If you want to leave the stack clean, you must specifically code a **drop** following **until**.

You may ask why MyForth does not just drop the top of stack after ending a loop. Although it isn't obvious from the simple examples given below, there are many cases when you need **t** for subsequent calculations. For example, when a loop terminates because **t** is non-zero, you may want to use the value of **t** that stopped the loop for subsequent calculations.

By explicitly coding a **drop**, you are only making the loop perform the same as it would if the **drop** was automatically compiled for you. However, if you need **t** after the loop terminates, you do not need to do anything special within the loop to preserve it and, after exiting, you don't have to do anything to restore it.

MyForth provides the following loop termination conditionals: **until, 0=until, again,** <literal> **=until,** <literal> **<until** and **until.**.

Using **again** as the loop conditional will form a loop that does not terminate. This is particularly useful in defining the startup Word (e.g., a Word named **go**) for an application that will be turnkeyed.

Note that most of these can operate on resources other than **t**.

Also note that **again** can operate over a range that exceeds an **sjmp**: it compiles **ajmp** or **ljmp**, as appropriate. The range of conditional jumps is shorter: they abort if out of range.

The following sections describe each of the remaining conditionals in more detail.

until and **0=until**

A **begin … until** loop operates as you might expect, looping until **t** is non-zero. This construct codes a **jz** instruction. Note that you may need to do a **drop** before exiting your definition, depending on what you want to do with the conditional value that terminated the loop.

If you want a loop that terminates when **t** is equal to zero, you can use a **begin … 0=until** construct. Here is an example:

```
: bloop   $10 #  begin  dup  .  1-  0=until  drop ;
```

This decompiles to:

```
---------- bloop
06A1  18           dec R0  (dup
06A1  F6           mov @R0,A  dup)
06A3  74 0A        mov A,#0A  #
06A5  18           dec R0  (dup
06A6  F6           mov @R0,A  dup)
06A7  91 BD        acall 048D  .
06A9  14           dec A  1-
06AA  70 F9        jnz 06A5  0=if
06AC  E6           mov A,@R0  (drop
06AD  08           inc R0  drop)
06AE  22           ret
```

From the above you can see that the loop termination value is kept in **t**. On entry, MyForth executes its usual **dup** to preserve **t**. Next, **t** is loaded with $0A by the sequence **$10 #** (**mov A,#0A**). Next a **dup** is executed so that **t** is not lost when **.** (dot) executes to emit the ASCII value of **t** back to the Host over the serial port.

The **1-** decrements **t** and the **0=until** checks to see if **t** is zero. The loop terminates when **t** is zero, leaving a **0** (the depleted loop counter) on the data stack. Finally, the **drop** removes the loop counter so that **t** will contain whatever was on the stack before **bloop** was executed.

=until

Here is an example of how **=until** might be used to count up in a loop:

```
: bloop1   0 #  begin  dup .  1+  $0A #  =until  drop ;
```

This will display **0 1 2 3 4 5 6 7 8 9** when it executes. We suggest that you define, download, execute and decompile this definition to become more familiar with the kind of code that will be compiled. You will see that it compiles code that is similar to that of the previous example, but it takes one more byte and it compiles a **cjne** instruction instead of **jnz**.

Please note the use of the **#** after the **$0A**; this is required for literals as well as stack items.

<until

If you code the above example using **<until** instead of **=until** as shown in the definition of **bloop2** below, the loop will terminate based on the value of **t** being less than minus 10. If you decompile **bloop2**, you will see that MyForth compiles code that is similar to that produced by the definition of **bloop1** above but two more bytes are required for the definition. Here is a definition using **<until**:

```
: bloop2   0 #  begin  dup  .  1-  -10 #  <until  drop  ;
```

When executed it will display **0 –1 –2 –3 –4 –5 –6 –7 –8 –9 –10** .

until. and **0=until.**

The **until.** loop terminator operates on a bit being set. The bit can be in any 8051 register, direct cell or port pin, including **t**. Here is an example:

```
: bloop4   1 #  begin  dup  .  2*'  5 .t until. drop ;
```

Note that the bit number, 5 in this example, does not need a **#** after it. This is because **until.** compiles the appropriate looping instruction using the data on the Host's stack; the number is not used by the Target at run time, only by the Host when it compiles the code into the Target's image. Executing **bloop4** will display the following: **1 2 4 8 16**.

Here is a more useful example:

```
#F8 constant SPI0CN
[ : .SPI0CN   SPI0CN + ; ]

:m wait-SPI   begin  7 .SPI0CN until. m;
```

The above example shows how you can loop until a bit is set in a special function register. It also illustrates that **until.** does not need to be used to check bits in **t** but can be used with other 8051 resources. Note how the bracketing in the definition of **SPI0CN** is used to access the Host to define the address to be used with **until.**.

Of course, **0=until.** operates in the opposite sense from **until.**, terminating when the specified bit is clear.

-until

The **–until** conditional terminates a loop when a cell goes negative (i.e., the most significant bit of the byte is set). Here is an example:

```
: bloop6   1 #  begin  dup  .  2*  -until  drop ;
```

This decompiles to:

```
---------- bloop6
0682  18             dec R0  (dup
0683  F6             mov @R0,A  dup)
0684  74 01          mov A,#01  #
0686  18             dec R0  (dup
0687  F6             mov @R0,A  dup)
0688  91 8D          acall 048D  .
068A  C3             clr C
068B  33             rlc A  2*
068C  30 E7 F7       jnb ACC.7,0686  if.
068F  E6             mov A,@R0  (drop
0690  08             inc R0  drop)
0691  22             ret
```

From the above will display **1 2 4 8 16 32 64**. Notice that this example uses **2*** instead of **2*'** used in **bloop4** above. If you compare the disassembly of the two definitions, you will see that the difference is that **2*** clears the carry bit before shifting. This was done in the example to ensure that you would get the same results as given here; if the carry had been set in a previous operation, the number sequence would be different.

Now, for the action of **–until**. Notice that the loop is formed by performing a jump based on whether or not bit 7 of **t** (the accumulator) is set. The intent of the minus in front of the "until" is to indicate 'negative' and should be thought of as "negative until."

Arithmetic and Logic

MyForth provides various operands to perform logical operations on stack items, direct cells and special function registers.

The following sections provide more detail, but key points to remember are:

- ***Operations on Special Function Registers, I/O ports and direct cells use logical Words ending with a "!" and do not require the "#" after them.***
- ***Conversely, operations on the stack are performed by Words that do not end in a "!" -- these do require the used of "#" to put their arguments on the Target's stack.***

This difference is because direct cell or port addresses are typically defined on the Host as constants (e.g., "$a4 constant PRT0CF"). Thus, when they are named in a definition their value is put on the Host's stack, not the Target's stack. The value on the Host's stack is then used by following Word to compile the appropriate instruction.

If this seems a bit confusing, read on. Hopefully the examples given in the following sections will make things clearer for you.

ior, **xor**, **ior!** and **xor!**

The **ior** Word performs a logical **or** operation. The “i” in the Word’s name stands for “inclusive” to distinguish it from **xor**, the “exclusive or” Word. Here is a simple example of “oring” two constants:

```
: ior1   $aa #  $55 #  ior ;
```

The **xor** Word is used in the same way as **ior**, but performs an “exclusive or” operation.

The use of **ior!** and **xor!** is similar to that of **ior** and **xor**, but the operand immediately preceding the instruction does not require a **#** after it. These Words are typically used with Special Function Registers (SFRs), direct cell addresses or port addresses. For example:

```
$a4 constant P0MDOUT
:m push-pull   $ff # P0MDOUT ior! m;
```

This example perhaps makes it clearer why SFRs are special cases: they are defined as ordinary constants in GForth and thus do not need to be put on the Target’s stack with **#**.

This example uses **ior!** in a macro that sets the outputs of a C8051F300 chip to push-pull. This macro would typically be used within an initialization Word.

and and **and!**

The **and** and **and!** Words perform a logical “and” operation and are used in the same way as the “or” Words.

+ and +’

Use **+** to add two numbers. If you need to add with carry, you can use **+’**. For example:

```
: addem  ( n1 n2 – n3 )   4 #  5 #  + ;
```

1+ and 1-

Use **1+** and **1-** to add or subtract one from **t**. These operations assume that **t** contains an 8-bit signed integer.

1u+ and 1u-

Use **1u+** and **1u-** to add or subtract one from the second item on the stack. These two Words should be thought of as “one under plus” and “one under minus.”

Here are two Words defined in **examples.fs** that you can try:

```
: incunder   ( n1 n2 – n3 n2)   1u+;
: decunder  ( n1 n2 – n3 n2)   1u- ;

22 #  33 #  .s  2> 33 22
incunder .s  2> 33 23
decunder .s  2> 33 22
```

Note that **1u+** and **1u-** are equivalent to **INC @R0** and **DEC @R0**.

negate and **invert**

Use **negate** to change **t** to a negative number. Here is an example:

```
: negate-example  ( – n )   44 #  5 # negate + ;
```

Use **invert** to invert all of the bits in **t**.

Because inverting all of the bits in a constant is a common operation for logical manipulation of port or SFR bits, MyForth also provides **~#**, as described elsewhere in this manual. You can use either **invert** or **~#**, depending on what you are doing.

You would typically use **invert** to manipulate a value that is already on the stack while **~#** is more useful (and efficient) if you just want to invert the bits in a constant prior to performing a logical operation.

As shown below, the two operators have the same stack effects but compile different code.

Here is an example

```
: invert-example  ( – n1 n2 )   5 ~#  5 # invert ;

invert-example  .s  2> -6  -6
```

This decompiles to:

```
---------- invert-example
076B  18             dec R0  (dup
076C  F6             mov @R0,A  dup)
076D  74 FA          mov A,#FA  #
076F  18             dec R0  (dup
0770  F6             mov @R0,A  dup)
0771  74 05          mov A, #05  #
0773  C3             cpl A  invert
0774  22             ret  ;
```

2*, **2*’**, **2/** and **2/’**

You can multiply or divide an item in **t** by two (left or right shift by one bit) using **2***and **2/**, respectively. If you need to use the carry bit, use **2*’** or **2/’**.

Here is an example using the carry bit:

```
: leftwith  ( - n )   [ setc ]  $c0  2*’  ;
```

In this example, the assembler is first used to set the carry bit, then the value $c0 is put in **t** and multiplied by two with carry (left shifted one bit with carry). The number $81 is left in **t** after the definition is executed.

|*

Use **|*** to multiply two 8-bit integers. The bar indicates that this definition is an inline macro that can be used within a colon definition or macro, but is not a callable Word. Here is an example:

```
: *  ( n1 n2 – n3)   3 #  5 #  |*  ;
```

After executing *****, 15 will be in **t**.

|um*

The inline definition **|um*** multiplies the two unsigned bytes on the stack, leaving the double precision (16 bit) result on stack with most significant byte in **t** and the least significant byte in the second stack cell.

As usual, the bar in the name indicates that **|um*** is a macro definition which can be compiled in a definition but is not callable. Of course, you can make **|um*** a callable definition by including it in a MyForth colon definition.

For example, this following is defined in **examples.fs** to make **|um*** a callable definition:

```
: um*  ( n1 n2 – n3 n4)   |um* ;
```

Executing **33 # 2 # um*** from the MyForth command line would put 0 in **t** and 66 under it in the second stack cell. The MyForth stack display would be: **2> 0 66**. Executing **ud.** after this operation would display **00066**.

|u/mod

The inline definition **|u/mod** divides the two unsigned bytes on the stack, leaving the quotient in **t** and the remainder in the second stack cell.

It is defined as an inline definition (indicated by the “bar”) because you may just want to use it to compile the code for **u/mod** within a definition without calling it. Of course, you can make it a callable definition by defining it as a colon definition as described in the previous example for **|um***.

The **debug.fs** file contains an example of how **|u/mod** can be defined as a callable definition and how it is used to define **u.** for interactive testing of Target definitions:

```
: space   32 # emit  ;
: digit   -10 # + -if -39 # + then 97 # + emit  ;
: u/mod   |u/mod   ;
\ Avoid leading zeroes in (u.)
: three   digit
: two   digit  digit ;
: (u.)   10 # u/mod  10 # u/mod  if three ; then drop if two ; then drop
          digit ;
: u.   (u.)  space ;
```

The above shows how **|u/mod** can be made callable, for example, to save memory when used multiple times in a definition. In **(u.)**, **u/mod** is used two times with a divisor of 10 to put three numbers on the stack.

Depending on the number to be converted, some of the values on the stack may be zero. Because it is not necessary to display these one or two leading zeroes, **(u.)** has logic to suppress them. The first “if” checks **t** to see if it is non-zero, indicating that there are three non-zero digits on the stack. In this case, it calls **three** which converts the number in **t** to an ASCII digit and emits it back to the Host. Because **three** is not terminated with a semicolon, a call to it falls through to **two** which converts two more digits.

If the first “if” finds that **t** is zero, then **t** is dropped and the second “if” checks the next stack item for zero. If it isn’t zero, then **two** is called to convert the remaining two digits. If the second number is zero, **t** is dropped and **digit** is called once to convert the single valid digit.

One final note: **three** and **two** in the definition of **(u.)** are followed by semicolons which normally compiles a **ret**. Disassembling **(u.)** reveals that these are optimized to jumps to eliminate redundant returns.

5

Assembler

Overview

This chapter describes how to use the assembler. Be forewarned, there isn't much to it. But, it provides most of what you need. And, if you require more, you can extend it.

The assembler is defined in the files named **misc8051.fs** listed in Appendix A. The assembler definitions are in the section starting with "---------- assembler." We suggest that you refer to this listing when reading this chapter.

Like the rest of MyForth, the assembler is simple. Instead of providing a full-blown RPN assembler, MyForth provides some basic assembler definitions that are sufficient to accomplish MyForth's mission: the efficient generation of 8051 code without the need to learn a complex system.

The assembler provides the essential tools you need; these are flexible enough, once thoroughly understood, to allow you to code transparently and efficiently.

Note that there is no "special" assembler file that must be loaded: the assembler definitions are included when you load **misc8051.fs**.

Assembly Definitions

In a typical Forth system, Code definitions consist of a named set of assembly language statements and/or macros that define a Forth Word. Code Words are normally preceded by a defining Word such as “code” and are often terminated with another special Word or sequence such as “end-code.”

Forget all that. In MyForth, there is no special defining sequence for Code definitions: Words defined in the Target vocabulary with a colon (Colon Words) are actually Code Words. MyForth Colon definitions can contain macros, bracketed assembly language sequences or references to other Colon Words.

The secret is now out: MyForth is essentially an 8051 macro assembler in disguise. MyForth definitions read like Forth but their secret mission is to efficiently compile 8051 assembly language definitions.

You may think that an RPN assembler and Code definitions are needed to efficiently compile 8051 assembly language. Although some useful assembler definitions are available in MyForth, efficient coding is provided as a natural feature of MyForth’s Colon and macro definitions. If this sounds strange, read on. Hopefully it will become clearer as the operation of the assembly definitions are explained.

For those attached to an assembler, this chapter explains how to use the MyForth assembler definitions. It also describes how to translate some standard 8051 syntax statements into assembly language statements.

In Line Assembly

Like using Code Words, most Forth programmers are accustomed to executing in-line assembly within their Colon definitions to improve efficiency or to directly access processor resources.

In MyForth, you normally code assembler definitions using the **[…]** sequence within a Colon definition in much the same way you would using special “in-line” encapsulation with a conventional Forth system.

However, because of MyForth’s ability to code definitions as macros, this is less necessary than in more conventional Forth implementations. As mentioned earlier, MyForth is primarily a macro assembler implemented within a Forth conceptual framework.

When you use the **[…]** sequence to include assembly language definitions, you are actually changing to the Host vocabulary and executing ordinary GForth Words that lay down assembly instructions in the Target’s image.

This is also what macro definitions do. In fact, it is often unnecessary to use assembler definitions within brackets, but this is commonly done to clarify the programmer’s intent and to avoid confusion.

Speaking confusion, there are several assembler definitions that have the same names as MyForth Words. These include **push**, **pop** and **swap** which are 8051 assembly language instructions and are also MyForth Words that manipulate the stack.

push and **pop**

In the context of the assembler, **push** and **pop** act on a register or direct cell.

Do not confuse the assembler versions with the Target versions: the Target versions act on the contents of the accumulator, t.

The section describing the return stack in the Compiler chapter shows how the assembler versions of **push** and **pop** are used to define the MyForth definitions of **push** and **pop** that apply to the data stack. As shown, the assembler versions of **push** and **pop** are applied specifically to **t** to push and pop the top of stack.

In the assembler, **push** and **pop** can be used with any direct cell.

set and **clr**

Use **set** and **clr** to set and clear bits within a register, port or direct cell. MyForth provides a number of operators beginning with “dot” to help specify bits within ports and registers such as **t**. For example, to set or clear bits in **t**, use **.t**. Here are two examples:

```
: set-example1  ( - n)   5 #   [ 1 .t set ]  ;
: set-example2  ( - n)   5 #     1 .t set  ;
```

In both cases, the result left in **t** is 7 (the bits are zero referenced). In the second example the “1 .t set” sequence does not need to be bracketed because **.t** and **set** do not exist in the Target vocabulary and are thus found in the Host vocabulary. The format shown in **set-example1** is perhaps preferable because it shows the intent of the coding somewhat better. Also, it is safer form to use when you are not sure whether or not a Word is defined in both vocabularies.

One final note: the “1” in the above just puts a 1 on the Host’s stack which is used by the following definitions to assemble the correct instructions; the 5 also goes on the Host stack, but the # following it assembles code that puts a literal in **t** when the definition is executed.

Pins and **Bits**

The following assembler Words are available to set, clear and toggle bits in ports:

setc, clrc (set and clear the carry bit)
set, clr (set and clear bits and port pins)
toggle, .P0, .P1, .P2, .P3 (port pins)

A previous section illustrated the use of **set** and **clr** with **t**. The syntax for setting and clearing port bits is similar to that shown in the previous example.

Here are examples showing how to use a port designation with **set** and **clr**:

```
:m enable    [ 3 .P2 clr ]  m;
:m disable   [ 3 .P2 set ] m;
```

Note that the assembly sequence is bracketed within the macro definition. This is because the macro compiling Word, **:m** establishes the Target vocabulary first in the search order so that macro definitions are compiled directly into the Target image.

The **[** before the assembler definitions establishes Forth first in the search order. ***This is because assembler definitions are GForth definitions that execute to do lay down code in the Target image.*** In the **enable** example, “3” puts a number on the Host’s stack and the “.P2 clr”, executed by the Host, assembles instructions in the Target image that set bit 3 in port 2. Here is a more detailed description of how the bracketed instruction sequence operates:

1. The left bracket establishes Forth as the first vocabulary in the search order,
2. “3 .P2” puts a bit address on the Host’s stack,
3. either $C2 (clrb) or $D2 (setb) is put on the Host’s stack,
4. the Target compiler is turned on and either $C2 or $D2 is placed in the Target image,
5. the byte compiled by “3 .P2” that was placed on the Host’s stack is written to the Target image,
6. the Target vocabulary is set as first vocabulary in the search order

To see what is compiled, use the decompiler (**see** command) to examine the definition.

Note that the brackets in the definitions of **enable** and **disable** are not really necessary and are coded more as comments than directives. This is because the bracketed items are not defined in the Target vocabulary. When the Host's dictionary is searched, they will be found and executed.

In the case of **enable**, a 3 will first be put on the Host's stack; all numbers not followed by a # will be put on the Host's stack. As previously noted, the **.P2** will convert the number on the Host's stack to a bit address appropriate for use by **clr** and then put it on the Host's stack. The **clr** instruction, defined on the Host, will execute to compile an instruction into the Target image.

All of this will be a bit bewildering at first, but the secret to MyForth's simplicity and efficiency is that you can use a few well-understood tools to achieve the results you want.

You should be prepared for some initial frustration and the frequent use of the decompiler to see what strange things you have asked the compiler to do.

However, the adjustment period is shorter than the one needed to understand a complex "kitchen sink" environment that provides a bewildering array of options that you are forced to wade through each time you try to do something.

Digging yet a little deeper, here are the definitions of **set** and **clr** contained in **misc8051.fs**:

```
[  \ these are assembler, not Target Forth
: set   $d2 ] , , [ ;
: clr   $c2 ] , , [ ;
```

The **[** ensures that the Host's compiler is used to define **set** and **clr**. The **$d2]** and **$c2]** sequences put bytes on the Host's stack and turn on the Target compiler by establishing it first in the vocabulary search order.

The first comma writes either a $d2 or $c2 byte in the Target image; the second comma writes the byte assembled by **3 .P2** from the Host's stack to the Target image.

The **[** ensures that Forth is established first in the vocabulary search order before exiting. Often this is not strictly necessary because the definitions will be executed within a sequence starting with **[**.

The above illustrates how MyForth manipulates the two vocabularies, Forth and Target, to control how things are compiled.

Finally, here is another example of how you might use **set** and **clr** in a definition:

```
:m SCLK 0 .P2 m;
:m (+P2.0)  SCLK set m;
```

or

```
: ~P2.0 [ SCLK toggle ] ;
```

Of course, you will not be able to see the decompiled code for **(+P2.0)** unless you put it in a definition. Remember that macros are not executable – they compile Target code when executed. Again, the brackets in the definition of **~P2.0** are not strictly necessary and serve mostly to indicate that they are assembler definitions.

mov

The most general of the MyForth assembler definitions is the **mov** instruction. It can be used in most of the ways that the **mov** instruction is used in an 8051 assembler. The following sections provide specific usage examples.

As mentioned previously, MyForth designates registers by their numbers, 0 through 7. Numbers above 7 are assumed to be direct cell addresses.

The following illustrate how to use the **mov** instruction to move data between registers and direct cells.

```
\ Move data in a direct cell to a register

:m direct-to-register  [ $15  3  mov ] m;
: testdr  ( - n)
  $15 # a!  22 # !          \ move 22 into direct cell $15
  direct-to-register        \ move contents of $15 to R3
  3 # a!  @ ;               \ put contents of R3 on the stack


\ Move data in a register to a direct cell

:m register-to-direct  [ 3 $15 mov ] m;
: testrd  ( - n)
  3 # a!  33 # !       \ move 33 into R3
  register-to-direct \ move contents of R3 to direct cell $15
  $15 # a!  @ ;        \ put contents of $15 on the stack


\ Move data in a direct cell to another direct cell

:m direct-to-direct  [ $15 $16 mov ] m;
: testdd  ( - n)
  $15 # a!  $a5 # !  \ move $a5 into direct cell $15
  direct-to-direct   \ move contents of $15 to direct cell $16
  $16 # a!  @ ;      \ put contents of $16 on the stack
```

Note that there is no assembler sequence shown for moving a literal into a direct cell. You can perform this operation using **#!** or **(#!)**, as described in the Compiler chapter. The Words #@ and **(#@)** are special cases that move data from a direct cell to **t**. If you decompile definitions using these Words, you will see that they compile **mov** instructions.

movbc and **movcb**

Use **movbc** to move a bit into the carry bit; use **movcb** to move a bit from carry into a bit address.

Here is an example:

```
:m @P2.3   3 .P2 movbc m;        \ move bit 3 of port 2 into carry
```

Note that, although the definition is a macro defined using the assembler, it is not bracketed. This is because **movbc** and **movcb** are not MyForth Words and thus are found in the Host vocabulary. When executed by the Host, like all assembler definitions, they compile assembly language statements into the Target image.

You could put the "3 .P2 movbc" sequence within brackets to emphasize that this is an assembly language definition.

[swap]

Use **[swap]** to swap nibbles in **t**: it is equivalent to the "swap A" assembly language statement. Because **[swap]** is not a Target Word, there is no need to include it within brackets unless your intent is to clarify your coding.

nop

Use **nop** to compile a "no operation" instruction in the Target's image. Here is a simple example to pulse a port pin:

```
:m pulse-P2.3   [ 3 .P2 set  nop nop nop nop nop  3 .P2 clr ]   m;
```

Note that the brackets are optional.

inc and **dec**

Use **inc** and **dec** to increment or decrement the contents of a direct cell or register. For example:

```
: bump-R7  ( - n)   $aa #  7 #!  7 inc  7 #@  ;
```

Executing **bump-R7** will leave $ab on the stack. Note that it is not necessary to bracket “7 inc” because inc is not a Target definition. Of course, you could code the definition as follows to emphasize the in-line assembly sequence:

```
: bump-R7  ( - n)   $aa #  7 #!  [ 7 inc ]  7 #@  ;
```

6

Boot Loader

Overview

The information in this chapter is primarily provided as a reference and for those wanting to understand more about the operation of MyForth. In normal operation you seldom need to concern yourself with the Boot Loader and its operation.

Purpose

The Boot Loader's sole purpose is to download the MyForth system definitions and your application code into the processor's flash memory. After it does that, it is no longer used.

Note that the Boot Loader does not execute commands interactively with the user. This function is performed by the combination of the tethering code on the Target and the Forth routines on the Host. The Boot Loader is invoked with the **d** command to download a compiled program.

Advantages

The primary advantage of using a Boot Loader is that it allows you to program the Target via the serial port instead of programming the Target using the Silicon Laboratories EC2 Serial Adapter and the Silicon Laboratories IDE. This simplifies the hardware needed for normal programming operations and also reduces the number of operations needed to program your chip.

Installation

Presently, for all chips except the C8051F12x chips, MyForth uses the AM Research Boot Loader and assumes it is located at $0000. In the future, MyForth will have its own Boot Loader that will be compatible with the AM Research Boot Loader and will also write a Boot Loader in each of its **chip.hex** (and **chip.bin**) program image files. The following describes this future configuration.

Installation of the MyForth Boot Loader for Silicon Laboratories chips requires the JTAG download hardware and software furnished with a development system from Silicon Laboratories or AM Research.

The AM Research Gadget modules support various Silicon Laboratories chips such as the C8051F300 and C8051F310. The chips on Gadget boards are furnished the Boot Loader already installed and you can use one of these with MyForth without any additional programming.

The AMR Development System also has a JTAG loader facility that will write a Boot Loader in any of the Gadget chips or any Silicon Laboratories chip connected to a 10-pin program adapter. How to do this is explained in the *AM Research 8051 Reference Manual* distributed with the AMR Development System.

Installation of a Boot Loader via the JTAG interface is adequately covered in the AM Research Development Manual and is not covered here.

Installation of the Boot Loader using a Silicon Laboratories Development System requires the following:

1. An EC-2 Serial Adapter connected to a 10-pin JTAG interface connector on a Silicon Laboratories Target Board (or wired to the JTAG pins of your own Target processor)
2. Use of the Intel HEX download function of the Silicon Laboratories Integrated Development Environment (IDE).

The EC-2 and IDE are furnished with Silicon Laboratories development systems. The use of the IDE’s Intel HEX download function is straightforward: just bring up the IDE and select the appropriate menu options.

The MyForth program compiler (executed with the **c** command), always produces two files: a binary image file and an Intel HEX file. The Intel HEX file, **chip.hex**, contains a complete program image, *including the MyForth Boot Loader*. To install the MyForth Boot Loader, is only necessary to download any MyForth program to your Target using the EC-2 and the IDE.

Once a program is downloaded to the chip via the JTAG interface, the Silicon Laboratories EC-2 and IDE are no longer needed. Thereafter, the Boot Loader will download your MyForth programs to the chip via the serial port.

Location

The Boot Loader resides in the first page of flash RAM starting at $0000 and is only used to load the compiled Target image via the serial port; it is not used thereafter.

Operation

As mentioned above, the Boot Loader is installed at location $0000. It consists of a very small amount of code that performs the following:

1. Sets up the serial port and communications rate
2. Re-maps the interrupt vectors
3. Checks to see if there is an active download request from the Host and, if so, it downloads code from the host over the serial port
4. After downloading or after a timeout period, it jumps to the MyForth system code and your application

The Boot Loader re-maps the interrupt vectors to Page 1. For most Silicon Laboratories chips, these will now start at location $200; for the C8051F12x chips this will be at $400 because of their larger page size.

If you disassemble your code starting at location $200 or $400, you will see a jump to Cold (the Cold Start vector). Other interrupt vectors may or may not follow this, depending on which ones your application needs. The MyForth system code starts immediately after the last interrupt vector. In some cases, if there are no other interrupts used, this will be immediately after the Cold Start vector.

Normally, the MyForth system code consists of definitions needed to implement the tether and a few definitions useful for interactive development (these are contained in **debug.fs** and **interactive.fs** – see the JOB file). Your application starts after the MyForth system code. For turnkeyed systems, the Cold Start vector points to the “go” definition for your application. For tethered application code, the Cold Start vector points to the tethering code (i.e., quit).

You can see all of the above by starting MyForth with the **c** command and then disassembling starting at Cold (i.e., **see cold**). Alternatively, you can use the **decode** command to disassemble at location $200 or $400 (e.g., **decode $0200**).

Because MyForth uses the AMR Boot Loader, examining the system image starting at location $0000 will show only $FF in all locations up to location $200 or $400. When MyForth has its own Boot Loader compiled in its binary or hex download files, you can examine the Boot Loader code by entering **decode $0000**.

As mentioned above, your application starts immediately after the MyForth system definitions. You can see where this is by defining a small test Word and then disassembling the resulting code with **see**. If you have compiled a turnkeyed application, you can examine your application code by entering **see go**.

The disassembler will show where your program starts. If your test program includes calls to routines such **emit**, you can see where these are located too.

Overhead

To some, the overhead of the Boot Loader and the debug definitions that MyForth may optionally load seems wasteful of processor resources. We feel that the small overhead is worthwhile, considering the following:

1. ***Interactive Test and Verification*** - The Forth system Words allow you to interactively exercise your code. Thus, you can interactively examine the operation of your new code without relying on a simulator. There is no substitute for the real machine, especially if you need to examine outputs at various pins or the behavior of connected hardware. Remember that you are seldom just verifying the operation of the chip; you are often verifying its interaction with connected hardware.

2. ***Code Reliability and Re-Use*** - All of the routines defined in the MyForth system are available for use by the programmer, either as Forth Words or as assembly code.

3. ***Reduced Program Size*** - You will notice that your programs do not grow very fast after a certain point because you are mostly re-using code that is already written and functioning reliably. This is especially true if you have factored your application properly so that useful functions are available for re-use. Thus, the MyForth code is not simply overhead you tolerate to get the benefits of interactivity; it is a reservoir of powerful routines that can make your programming much simpler. For non-trivial applications re-use of code can significantly reduce the size of your application; for small programs, program size is not an issue.

7

Tethered Target

Overview

This chapter describes the communications tether between the Host and Target processors. Because the tether provides MyForth's interactive Forth environment, the following is provided for those who wish to know more about it.

However, it is not necessary to know how the Tether works to perform normal programming operations.

The source code for the Target interpreter (tethered interpreter) is contained in **tether.fs**. The required definitions are few and deceptively simple; here is the entire listing:

```
] \ Target Forth
: emit begin 1 .SCON until. 1 .SCON clr SBUF #! ;
: key begin 0 .SCON until. 0 .SCON clr SBUF #@ ;
: ok 7 # emit ;
: number ok key ;
: execute swap push push ;
: quit key emit key key execute ok quit ;
```

Before examining the above, there is some basic information about tethering that may help you understand it.

Basic Operation

The Host connects to the Target via a serial port and interacts with it using a simple protocol that tells the Target what code to execute. Such a system is usually called a "Tethered" Forth because the Target is tethered to the Host via a communications link.

With a Tethered Forth, the Host PC performs most of the Target interpreter's work. Although it appears to the user that the Forth system is executing on the Target, most commands are executing on the Host which tells the Target what to execute. Thus, the Host only communicates to the Target when it needs it to execute some code.

To implement a tethered Forth, the Target only needs to be able to execute code at a specified address. Because of the simplicity of this requirement, the code overhead on the Target is minimal.

execute

The number and function of the commands in a tethered Forth can vary. The MyForth Target uses only one command, **execute**, to do the Tether's heavy lifting. This command takes two bytes on the Target's data stack and pushes them on the Targets return stack in the proper order.

In the code given at the start of the chapter, the semicolon at the end of **execute** compiles a **ret** instruction, as usual. Thus, **execute**, when it completes, "returns" to the address just pushed on the return stack. This performs the equivalent of a jump (not a call) to the specified address. After the code at the address is executed, it executes a **ret**, thereby returning to the code following **execute**.

In the above example, where **execute** is contained in the definition of **quit**, the **ok** that follows **execute** will be called. This may seem a bit confusing or weird, but is worth understanding.

quit

To get the two execution bytes on the stack, the Target sits in an endless loop, defined by **quit**, that looks for execution addresses transmitted from the Host over the serial link. When these are received, **execute** jumps to the specified address.

Examining the code for **quit**, the sequence “key emit” simply waits for a byte to arrive from the Host on the serial link and then echos it back. This signals the Host that the Target is listening for the next address to execute.

When the Host receives the echo, it sends the two address bytes. The sequence “key key execute” gets the two bytes sent by the Host and jumps to them, as described above.

The “ok**”** signals the Host that the Target has executed the code by sending it a “7.” The **quit** at the end of **quit** is a tail recursive call, returning execution back to the beginning of the **quit** code (i.e., it is an endless loop).

The definitions of **key** and **emit** use the standard 8051 serial port flags and registers to wait for a character (**key**) or send a character (**emit**).

8

Standalone Target

Overview

MyForth allows you to install a Forth system on the Target and interact with it with a dumb terminal. This Standalone Target has the basic features of a Forth system including an interpreter, a dictionary and stacks. With a Standalone Target, you can communicate with your application without having a MyForth system installed on a Host PC.

Thus, a Standalone Target is useful for interacting with a system over a serial port when a tethered interaction is not practical. These applications would include control, monitoring and testing of a deployed target.

Installation

To install a Standalone Target, edit **config.fs** in your project directory so that the **tethered** constant is false (zero):

```
false constant tethered   \ Standalone Target
```

After making the above change, compile and download your application using **d**, as you would normally do for a tethered application. Afterward, you can interact with your application using a dumb terminal at the same baud rate that you used to download your application (e.g., 9600 or 38.4K baud).

Operation

Dumb Terminal

To make interactive testing of a Standalone application easier, MyForth provides a dumb terminal that can be executed from a MyForth command line. This terminal is optional and is loaded in **loader.fs**. Of course, you can use another terminal program, such as Hyperterm, to exercise the Target.

To use MyForth's dumb terminal, simply type **dumb** at the MyForth prompt. You can escape from the dumb terminal with Ctl-C.

Stack

Entering numbers on the Standalone's stack does not require them to be followed by a #. This is because there is no need to distinguish between Host and Target stack operations.

Because of the limited size of the Standalone's terminal input buffer (tib), numbers and Words are immediately interpreted after you enter a space or carriage return. Also, remember that Standalone numbers are only 8 bits wide. All numbers are assumed to be in decimal; you <u>cannot</u> enter Hex numbers by prefacing them with a "$."

To display the Target's stack, type **.s**. The Word **depth** is available to check the current stack depth.

Words

Forth Words that you have defined in MyForth can be executed by entering them on the dumb terminal. However, because of the limited size of the Target's terminal input buffer, Words must be entered one at a time (you cannot have multiple entries on a line). Target dictionary entries are stored as the first three characters of the Word and a count. Thus, it is possible to have name conflicts. This is usually not a problem, but duplicate names are not flagged as errors: you must avoid this on your own.

When entering a Word to be executed, pressing the backspace key will abort the current entry and require you to re-enter the entire Word (or number).

Interpreter

The following provides a very basic description of how the Standalone interpreter operates, including the structure of the Target's dictionary.

The Standalone Target's code is contained in **standalone.fs** and is also given on the following page. The descriptions in this chapter are based on this listing.

Basic Definitions

The listing starts off by defining some basic building blocks for character I/O such as **key**, **emit**, **echo**, **space**, **cr** and **ok**. If you are familiar with Forth, these Words are no great mystery.

Similarly **2dup**, **clip**, **min** and **max** manipulate and clip stack items. These will not be discussed in detail either, but note that they are all defined in terms of the MyForth constructs covered in other parts of this manual. In particular, note the use of the **until.** and **–if** in the definition of **key**, **emit** and **clip**.

The three Words to manipulate that data pointer are worthy of note only in that they are defined with versions beginning with a "bar." For example, **p**, which puts the low and high bytes of the data pointer on the stack, is defined with **|p**.

In MyForth, whenever a Word begins with a "bar", it indicates that this is an "inline" or macro definition. These "barred" definitions just lay down instructions and are often used within a normal Forth Word to make them callable.

The **depth**, **huh?** and **?stack** Words perform basic stack checking and abort functions and do not need much explanation. Note, however, that the stack pointer is **s**.

The start of **tib**, the terminal input buffer, is defined with a constant. It starts just after the stack and address pointers, **S** and **A**.

You may have observed that some Words are defined with"-:" instead of ":". These Words are callable by other Words, but their headers are not compiled on the Target and thus they cannot be executed interactively from a terminal. Of course, these Words have the advantage of not taking up any dictionary space on the Target.

```
] \ Target Forth
: emit         begin 1 .SCON until. 1 .SCON clr SBUF #! ;
: key          begin 0 .SCON until. 0 .SCON clr SBUF #@ ;
-: echo        dup emit ;
: space        BL # emit ;
: cr           13 # emit 10 # emit ;
-: ok          space [ char o ] # emit [ char k ] # emit cr ;
: 2dup         (over) (over) ;
: min          2dup swap
-: clip        negate + -if push swap pop then 2drop ;
: max          2dup clip ;
: p            |p ;
: @p           |@p ;
: @p+          |@p+ ;
\ : depth [ SP0 -2 + ] # S #@ negate + ;
: depth        S #@ invert ;
-: huh?        [ char ? ] # emit cr reset ;
-: ?stack      depth -if huh? ; then drop ;
2 constant tib  \ begins after S, and A.
-: match ( ? - ?)      @+ @p+ xor ior ;  \ 0 if still a match.
-: word        0 # tib # a! match match match match ;
-: ?digit
        [ char 0 negate ] # + -if huh? then -10 # + +if huh? then  10 # + ;
-: number
        tib # a! 0 # @+ 3 # min begin swap 10 # (*) @+ ?digit + swap 1-
        0=until drop ;
-: find
        @p if drop word if drop p+ p+ find ; then invert ; then drop  0 #
        here constant dict  \ Patch this later with real dictionary.
-: dictionary          0 ##p! ;
-: interpret
        p push push a push dictionary find if drop @p+ @p pop a! pop
        pop p! push push ; then drop number pop a! pop pop p! ;
-: tib! ( c)   a push tib #@ 1+ tib #! tib # dup a! @ + 6 # min a! ! pop a! ;
-: 0tib        tib # dup a! 0 # dup !+ dup !+ dup !+ dup !+ ! a! ;
-: query       0tib
-: back
        key 8 # =if drop cr query ; then BL # max echo BL # xor if
        BL # xor tib! back ; then drop ;
-: quit        query interpret ?stack ok quit ;
```

Quit

The definition of **quit** is somewhat similar to that of more conventional Forths in that it is an infinite loop, querying for user input, interpreting or aborting as appropriate and then returning for more.

The first thing to note about quit is that it is defined using efficient tail recursion instead of as a **begin … again** loop. This illustrates another way that MyForth allows you to loop.

The first thing that **quit** does may seem a bit strange: it calls **query**, which simply clears the **tib** and falls through to the next definition (i.e., there is no semicolon to compile a return).

The mystery of **query** is solved by examining the following definition, **back**. One function of back is to restart **tib** when it encounters a backspace. If **back** sees a backspace, it just jumps back to **query** to start over. Note: pressing backspace does not remove characters from the **tib**; it aborts the current entry and requires the user to re-enter the entire Word. This is much simpler than keeping track of backspaces.

Assuming that **back** gets a valid character, it next checks to see if a blank has been entered. It does this by performing an **xor** with the entry. If a blank is entered, then **back** returns; its job is done and some characters are in the **tib**, ready to be interpreted. If the user has not entered a blank to signal the end of an entry, blank performs another **xor** to recover the entered character, stores it in **tib** and returns for more characters.

Following sections describe the processing of the characters captured at **tib**.

After the input characters are processed, the result is one of the following: 1. A jump to the Word at **tib**, 2. A number put on the Target's stack resulting from the execution of **number**, or 3. A stack error. The "found" and "number" actions are discussed below.

If there is a stack error (e.g., as a result of processing by **number**), the stack pointer will be positive. The execution of **depth** fetches the stack pointer and inverts it to form a flag for **–if**. Thus, a stack error will result in the execution of **huh?**. This Word simply emits a question mark and jumps to the reset vector. The reset vector is defined as location zero at the end of **misc8051.fs**.

If there are no stack errors (i.e., a Word was executed or a number was put on the Target's stack), then an "ok" is sent to the Host to signal successful execution and tail recursive **quit** is executed to continue the indefinite quit loop.

Interpret

Interpret looks a little intimidating, but is really quite simple. The first few Words just save the data pointer (p) and address register (a) prior to loading the data pointer with the address of the start of the Target's **dictionary**.

Note that the data pointer is apparently zeroed. But, the address of the start of the dictionary is patched after your application is compiled. To see how this is done, refer to the **job.fs** file. At the end of it you will see the following:

```
tethered [if]  \ For interactive testing, entering numbers.
        :m # number emit-s m;
        :m ## [ dup 8 rshift $ff and swap $ff and ] # # m;
[else]  headers ] here [ dict org heads ##p! org ]
[then]
```

The part we are interested is the **[else]** condition that is executed if the application is standalone (i.e., not tethered). The "headers**"** copies the dictionary from its separate address space over to the end of the dictionary, setting the value of **heads** to point to the beginning of the dictionary. Then "here**"** is invoked put the Target's dictionary pointer on the stack for later restoration. The "dict org" phrase sets the Target image compilation address to the address at which we previously compiled the "0 ##p!" instruction. The "heads ##p!" recompiles (and over writes) the instruction at that address using the correct address of the start of the dictionary. Finally, "org" restores the Target's compilation pointer to the address that it had before performing the dictionary patch.

The **interpret** Word next uses **find** to find a potential dictionary entry. The operation of **find** is explained below. Assuming that a dictionary entry is found, DPTR (p) points to the byte immediately following the found header. When **interpret** executes **@p+**, it fetches the first byte at the address pointed to by DPTR (p) and also increments the data pointer to the following byte. This byte is the second byte of the execution address for the definition just found. The **@p** instruction gets this byte and puts it on the stack together with the previous byte; at this point the execution address of the found Word is on the Target's stack.

Next, the phrase "pop a! pop pop p!" restores the address and data pointers that were saved at the start of the definition. Last, **interpret** pushes the execution address of the found Word on the return stack and executes **;**. This compiles a **ret** instruction that will result in a jump to the execution code for the found definition.

If a match is not found in the dictionary for the characters at **tib**, then **number** is executed to try to convert the characters to a number. Following this attempt, the address and data pointers are restored.

find

The **find** Word searches through the Target code until it finds a non-zero value. During the search, the data pointer is incremented. When it finds a potential dictionary entry, it drops the value and proceeds to **word**. First, **word** puts a zero on the stack to act as a “found” flag and loads the address of **tib** into the address (**a**) register; this will be used to fetch characters from **tib** that will be used by **match**.

To see if the dictionary candidate matches the contents of **tib**, **match** performs an **xor**; if the items match, the value will be zero. This value is “ored” into the top of stack to maintain the “found” flag. The match is performed four times to coincide with the four-cell size of a dictionary entry. When all four matches are complete, the “found” flag (top of stack) will be zero if a match was found or non-zero if there was no match.

9

Examples

Overview

The examples described in the following sections are included with the MyForth distribution. They show typical MyForth applications and show how various Words and macros are used.

The following also explains how the various Words, macros and constructs in the application work. The explanations can be a bit long-winded for more experienced users but are primarily intended for neophytes taking their first look at MyForth application code.

The first application, an LCD driver for the C8051F300 chip, provides more detail than the second, a random sequence generator for the C8051F120 chip.

Thus, if you are trying to learn about MyForth, read the first example and then proceed to the second. If, however, you are just reviewing typical applications, you can start with either example or with the source code itself.

If you elect to start with the source code, the detailed coverage in the examples may be able to explain coding that you don’t understand.

LCD

Project Description

The LCD application is contained in the **..\myforth\projects\lcd** directory and is coded for a Silicon Laboratories (SL) C8051F300 chip. The code is very basic and you will undoubtedly want to expand it for LCD display with your application.

The code assumes you will use the nibble mode interface to the chip. The code was converted from a working application in AMR Forth. The AMR application file, **amrlcd.fs**, is included in the LCD project directory.

Hardware

The application hardware was prototyped on an AM Research Development Board with a 300 Gadget plugged into the processor slot.

The connections to the Gadget were made to a 16-position single row IDC pin header, using wire wrap wire. The LCD module is a standard assembly with a 16-pin in line connector at one edge. An IDC socket header was soldered to the pads at the edge of the LCD assembly so that it mates with the header on the AMR development board.

An alternate approach to prototyping the hardware would be to use an SL Target Board for the processor and its serial interface to the Host. The IDC header interface to the LCD module can be wired on an AB1 Applications Prototyping board that mates with the Target Board via a standard 41612 DIN connector.

If you use this approach, you will have to download a Boot Loader to the Target Board using the EC2 Serial Adapter and the SL IDE. Alternatively, you can use the AMR Development board to download to the SL Target Board, as described in the AMR Manual. Yes, this is a bit like carrying coals to Newcastle, but you may want to use a processor not supported by AMR to drive your LCD.

To establish a Boot Loader, you can download the **chip.hex** file from the LCD project directory. This image file not only contains the Boot Loader, but also the working LCD application.

Note that all MyForth image files contain not only your application code, but also a Boot Loader image.

The source code for the LCD application is contained in **lcd.fs** in the LCD project directory. The LCD pin assignments are given in the comments at the beginning of the file, as follows:

```
\ lcd.fs

\ Nibble mode LCD driver for C8051F300 -- 22Aug06 cws/rjn
\ based on working driver written in AMR Forth, amrlcd.fs

0 [if]

LCD   PORT      300   FUNCTION
      PIN       PIN
---   -------   ----- ------------------------------------
 1    ----      11    GND
 2    ----            +5 Volts (used 78L05 on Gadget MB)
 3    ----            Contrast Voltage 0-5 Volts (10K pot)
 4    P0.7      10    RS - Instruction Register Select
 5    GND       11    R/W - H=READ L=WRITE Registers (GND)
 6    P0.6      9     E - Enable  P0.6
 7    GND       11    byte DB0
 8    GND       11    byte DB1
 9    GND       11    byte DB2
10    GND       11    byte DB3
11    P0.0      1     byte DB4 - nibble DB0
12    P0.1      2     byte DB5 - nibble DB1
13    P0.2      4     byte DB6 - nibble DB2
14    P0.3      5     byte DB7 - nibble DB3

[then]
```

Navigation

Although the following sections explain each part of the LCD application using actual code from the application source, you may want to examine the code directly. If you haven’t used Vim to navigate source code files before, this will help you become more proficient in doing it.

To examine the application code, we suggest you start by bringing up a Command Window, and changing to the **..\MyForth\projects\lcd** directory and entering: **e job.fs**. This will bring up Gvim and display the **job.fs** file. The “e” command is not covered elsewhere in the manual because it is just a batch file shortcut to typing “gvim <filename>” when editing a file.

Another way to start up Vim editing **job.fs**, is to enter: **e –t job.fs**. This brings up Vim with the “-t” option. This option specifies that the following Word should be looked up in the tags file and the editor started up with the file that contains it.

In this case, the definition of the cold start vector is used because it is contained in the **job.fs** file. Although in this example it is just as easy to start up without using the “-t” option, you will often want to start up this way because you will be interested in changing or examining a particular Word or macro.

Once you have started up in the Job file, place the cursor over the first character in “lcd.fs” and type “gf” (go file). This will bring up the **lcd.fs** file that you will be examining. To go back to the Job file, just enter “Ctrl-6” (we like to think of it as Ctrl-^ for ‘go up to the previous file’). We recommend starting up in the **job.fs** file and navigating to the file you want to edit or starting up with the “–t” option if you just want to change or examine a particular definition or code grouping.

Inclusion

Nobody wants to be left out. Even relatively minor coding players such as files defining Special Function Registers need to be included.

Before progressing from the Job file, note that, in addition to including the LCD application, the Job file also “includes” a file that specifies the Special Function Registers.

This is typically how you might want to build your application, with minor players being included in the Job file along with your main application file (e.g., **lcd.fs** or **main.fs**). There is a comment in the **lcd.fs** source that notes that the SFRs are included elsewhere. You may recall that the SFR definitions are copied to your project director when you configure it for a specific processor (i.e., by copying the contents of the 300 directory to the LCD directory). In fact, you can easily tell what processor is being used for a project by looking for the SFR file. In the case of the LCD project, the SFR file is **SFR300.fs**.

Here is the comment about the inclusion of the SFR file as it appears in the LCD source:

```
\ SFR definitions are included in job.fs
```

Observe that comments appear in blue.

I/O Configuration

The output pins that will drive the LCD must be set for push-pull operation.

Examining the source below, the first step in doing this is to execute the "left bracket" Word so the Target's vocabulary is searched first. This is a bit of "belt and suspenders" programming because the following definitions are macros and the ":m" defining Word executes a "]" itself. To see how **[**, **:m** and **]** are defined, look at the top of **compile.fs**.

Here is the code to define the I/O:

```
] \ Target Forth

\ ---------- I/O
\ set all outputs as push-pull
:m push-pull        $FF # P0MDOUT ior! m;

:m pins        P0 m;
:m dirs        P0MDOUT m;

:m .E          6 .P0 m;
:m .RS         7 .P0 m;
: instruction [ .RS clr ] ;
: data         [ .RS set ] ;
```

The first thing to observe about the I/O configuration is how the SFR constant, **P0MDOUT**, is used with **ior!**. The **#** following $FF compiles code in the Target image that will put $FF on the Target's stack when **push-pull** is executed.

The action of **P0MDOUT** is just that of an ordinary GForth constant: when it appears in the source, the appropriate constant is put on GForth's stack. The **ior!** consumes this constant from the GForth stack and lays down code in the Target's image that will use the $FF on the Target's stack at run time to "inclusively or" bits into **P0MDOUT**. Whew! Got that?

You may wonder why **P0MDOUT** does not need to be treated specially (i.e., with a "[…]" pair). After all, it is included within a macro that always sets the default action to compile into the Target image. Special treatment is not required because **P0MDOUT** is not defined in the Target's dictionary, so the vocabulary search "falls through" to GForth, where **P0MDOUT** is found as a constant and put on the GForth stack.

This illustrates an important point about examining GForth applications, particularly those written by Charley Shattuck. The use of brackets may not be as you would expect. This is because, once thoroughly understood, you can eliminate some of the unnecessary or explicit context switching.

The use of **P0MDOUT** is one case: it is understood (by Charley, at least) that any constant that is not followed by a # will just be put on the GForth stack. Similarly, any Word not defined in MyForth will "fall through" MyForth's vocabulary search and be executed by GForth.

This is the case for almost all assembler Words, with the notable exception of **push** and **pop**: assembler Words are normally GForth definitions that execute to compile code in the Target's image. The **ior!** Word is another example of this type of Word that GForth executes to compile code in the Target image.

Now that you think you just might understand **push-pull**, look at the macro definition for **dirs**. More confusion: here is a macro with the sole purpose of putting something on the GForth stack. This illustrates how you can use the macro's naming capability to make your code more readable. Remember, if you are targeting the LCD code for another processor, **dirs** may refer to a different SFR.

The definitions of **.E** and **.RS** are reasonably straightforward: they use the "dot port" Words to create macros that refer to specific pins on Port 0.

Finally, we come to some good ol' MyForth colon definitions! The instruction and data Words use the left and right bracket to contain assembly language statements. As mentioned earlier in the manual, you will sometimes see assembly language statements that are not enclosed within brackets because the assembly language Words are not defined in MyForth and will "fall through" to GForth for execution.

You may now wonder how it is possible to know which Words are defined where. Often, you will "just know" from experience. But, the best way to determine how to code is to examine the instructions assembled by your definition.

This is the MyForth way: a few simple constructs and lots of code examination. At first, be prepared to look at a lot of code: it will soon become clear what does what, how it does it and when. After some experience examining code, you too will "just know."

Delays

Thankfully, there isn't as much to say about delays as there was about I/O.

The **us** and **ms** delay Words are pretty straightforward, using the **#for … #next** looping construct to perform nested loops.

Note that the register to be used for looping must be specified before both **#for** and its matching **#next**. Also note that, in **us**, there is a "6 #" just preceding the "6 #for" specification. This is just a coincidence: the first 6 is just the number of times to loop and the 6 before **#for** is the register we are using for the loop.

The strobe Word uses the assembler to set and clear bits with appropriate delays in between. The delays were checked out on a scope but are valid only for a 300 chip. A separate project directory, **delays**, is included with MyForth that allows you to check the timing of the delay loops. The **delays.fs** file is also included in the LCD project directory for reference.

```
\ ---------- delays
: us  ( n - )
        7 #for
                6 #  6 #for 6 #next
        7 #next ;

: ms  ( n - )
        7 #for
                100 # 6 #for
                        81 #  5 #for 5 #next
                6 #next
        7 #next ;

\ 160 us may be ok at half the value given
: strobe   [ .E set ] 100 # us [ .E clr ]  160 # us  160 # us ;
```

LCD Words

There is not much new to discuss in the definition of the Words that actually control the LCD. But, it is worth noting the “[$c4] ,” sequence on second line of the **lcd** definition. This is an example of how you can just “comma in” an 8051 instruction that is needed, but not used frequently enough to justify inclusion in MyForth.

In this case, the instruction, $c4, swaps nibbles in **t** (the accumulator). This is handy for a nibble-mode LCD driver, but not frequently used otherwise. Actually, MyForth now provides the **[swap]** assembly language instruction to do this (popular demand, no doubt). The application code has been left as originally coded to illustrate how instructions can be added to your code.

```
\ ---------- lcd words
\ $c4 swaps nibbles in the accumulator in one cycle
: lcd  ( c - )
        $b0 # pins and!      \ output high nibble first
        dup $f0 # and [ $c4 ] ,  pins ior!
        strobe
        $b0 # pins and!      \ now output the low nibble
        $0f # and pins ior!
        strobe ;

: clear-lcd  instruction 1 # lcd data 100 # ms ;

: init-lcd
\    instruction
        $30 # pins #!        \ RS=0, instruction mode
        $cf # dirs ior!      \ configure pins as outputs
        30 # ms              \ power on delay
        $03 # pins #!        \ initialization pattern
        strobe 10 # ms
        strobe
        strobe
        $02 # pins #!
        strobe 10 # ms
        $28 # lcd 10 # ms
        $0e # lcd
        $01 # lcd 10 # ms
        $02 # lcd
        data  10 # ms  ;

: init  push-pull init-lcd ;
```

Strings

The following describes how to output a string to the LCD using the LCD Words just discussed. The definitions are a bit tricky, but not all that bad.

First, **lcd-type** takes an address that contains a counted string and outputs it to the LCD using **lcd**. The **p!** stores the address into DPTR and the **@p+** gets the count byte, putting it into **t**, and increments DPTR.

The "begin … 1 - 0=until drop" loop fetches a byte, increments DPTR (**p**), outputs the byte to the LCD (using **lcd**), subtracts one from the count on the stack and checks for loop termination (when it decrements to zero). After loop termination, the loop counter is dropped (this is a ~~peculiarity~~ feature of MyForth).

The macro definition for " appears a bit frightening, but isn't all that bad. First, **"** parses the text that follows it until it comes to a double quote. When finished parsing, the location of the string is at **here**. That address is put on the GForth stack and then **there**, the current compilation address in the Target's image, is put on GForth's stack. Then the two addresses are used to **place** the string in the Target's image.

Next, **here** is executed to put the address of the just-parsed string on the Gforth stack – for now, just remember that it is on the stack. Next, **there** is executed to put the address of the string in the Target image on the GForth stack. This is then used to advance the Target image pointer past the string so that following definitions don't overwrite it. Thus, when **"** finishes executing, it leaves a pointer to the string (in GForth's space) on the stack.

Now on to **string**: it executes on the Target, putting the current execution address on the Target's stack, adjusting the bytes for correct byte order and passing the baton to **lcd-type** for string output.

```
\ ---------- Strings.
: lcd-type ( da) p! @p+ begin @p+ lcd 1- 0=until drop ;

:m " 34 parse here there place here there [ c@ 1 + ] allot m;

: string pop pop swap lcd-type ;

: greet string " It's MyForth!"

\ Example: init greet
```

Random Sequence Generator

Project Description

The PSR project directory contains an application for the SL C8051F120 chip to simulate a 32-bit shift register with exclusive or feedback to generate a pseudo-random sequence. The bits are selected to produce a maximum length cycle (i.e., using bits 18 and 31 will produce a repeating cycle 2^{32}-1 long). This code can run on just about any of the SL chips (or other 8051 processors), with appropriate changes to the SFR and I/O assignments. However, the 120 chip was selected for this application because of its high speed.

A Microsoft Word document, **PSR.doc**, is included in the PSR project directory. It provides greater detail about the application and oscilloscope photos of the output that is generated from the 120 chip.

Hardware

The application hardware simply consists of a SL Target Board for the 120 chip with an oscilloscope attached to pin 6 of Port 1. This can be easily accessed via the IDC pin headers on the SL Target Board.

As noted above for the LCD project, you may have to download a Boot Loader to the Target Board using the EC2 Serial Adapter and Silicon Laboratories IDE.

The file to download to the 120 is the **chip.hex** file contained in the PSR project directory. As with the LCD application, this file not only contains the Boot Loader, but also the working PSR application.

Note that all MyForth image files contain not only your application code, but also a Boot Loader image.

Inclusion

As noted in the LCD description, the Job file “includes” some auxiliary files, **preamble.fs** and **io.fs**. The **preamble.fs** file defines 120 SFRs and includes code to start the chip running at 98 MHz. The **io.fs** file initializes the 120’s crossbar. The remainder of the application is discussed below.

I/O

Pin 6 of Port 1 is used to flash an LED. This assignment was chosen because the SL Target Board provides an indicator LED that can be driven from this pin and it is readily available for observation on a scope. Pin 7 of Port 1 is a monitoring output for the sequence generator – when turned on, it toggles once every time through the **psr** routine, thus providing a measurement of how fast the register is being shifted. The macros **outbit** and **clue** provide names for these two bits. The macros **+clue** and **–clue** turn the clue bit on and off.

You can see that these definitions do not need to use brackets to set up compilation for assembly definitions **set** and **clr** – they are defined in GForth and fall through the Target vocabulary search. Refer to the section on **set** and **clr** in the Assembler chapter for more detail.

```
\ psr.fs

:m outbit     6 .P1 m;  \ LED
:m clue       7 .P1 m;  \ For timing

:m +clue clue set m; :m -clue clue clr m;
\ :m +clue m; :m -clue m; \ disappear.
```

Target Byte Allocation

The four bytes used to form the 32-bit shift register are allocated with **cpuHERE**, which contains the address of the current pointer to the Target's RAM. Thus, **sequence** points to the four cells to be used as a 32-bit shift register. The **cpuALLOT** Word just moves the allocation pointer past the four bytes: they are initialized elsewhere.

The comments that follow the allocation show the organization of the register, including the address of the four bytes and the location of the feedback bits.

```
cpuHERE constant sequence 4 cpuALLOT
\ XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX
\ ^ bit 31     ^ bit 18
\ ^ sequence        ^ sequence+2
```

Seeding

The **?seed** Word checks to see of all of the bits in the register are clear and, if not, re-seeds it with a non-zero constant. The prevents shift register “lock-up.”

In **?seed**, the direct cell address of the MSB of the register, **sequence**, is put on the stack with **#** in the usual way and then stored in the address register (NOT the accumulator!) with **a!**.

Next, $aa is put in **t**. The following repeated sequence of “dup !+” stores $aa into the four **sequence** bytes indirectly through **a**. The **!+** Word not only stores the top of stack into the next **sequence** cell but also increments **a** so that it points to the next **sequence** byte. Perhaps you can now appreciate the use of **a** and its auto-incrementing operators to fetch and store data from or to sequential bytes.

As you can see (by executing **see ?seed**), the operation of **?seed** is very efficient.

```
\ If all bits are clear, reseed with $aaaaaaaa.
: ?seed
        sequence # a! $aa #
        dup !+ dup !+ dup !+ ! ;
```

Shifting

The routine that does the “heavy lifting” for the PSR algorithm is **psr**. Per the comment, it shifts the register one bit and handles the “exclusive or” operation on the two feedback bits. The comments explains most of the code adequately. You may have noticed the liberal use of brackets in the definition of this macro. They are primarily used to allow the calculation of addresses in GForth that are subsequently used to compile Target code that performs operations on these addresses (e.g., to compile Target code that fetch or store data relative to these addresses).

The first example of this is the phrase “[sequence 1 +] #@” that calculates the address of the second byte and puts its contents in **t**. The “#” in “#@” indicates that the byte address of **sequence+1** on the GForth stack is treated as if it were followed by a **#**. The code in **#@**, when it executes, takes a value off of GForth’s stack and compiles the appropriate Target code to put the contents of **sequence+1** on the Target’s stack (in **t**, actually).

The next bracketed sequence contains assembly instructions to move bit 2 of **t**, which now contains the contents of **sequence+1**, into carry. Then, carry is moved into bit 7 of **t**. Now everything is set up to “xor” bit 7 of **t** with bit 31 of the MSB and shift the result into carry. Note that **2*’** is used to left shift because carry must be preserved. The “xored” result is output to **outbit** for observation on the scope.

The following code shifts the “xored” bit into the beginning of the register and shifts all of the existing register bits toward the MSB. Again, note the use of **2*’** to ensure the carry is preserved so that it can be used to shift a bit into the LSB of the next byte in the sequence. The “parened” versions of **#@** and **#!** are used so that execution isn’t slowed down with unnecessary stack manipulation, as noted below.

```
\ Shift once with feedback from bits 18 and 31.
:m psr       +clue
      [ sequence 1 + ] #@            \ Get bit 18.
      [ 2 .T movbc  7 .T movcb ]     \ Move it to bit 7 of TOS.
      sequence #@ xor                \ xor bits 31 and 18.
      2*'                            \ Move xored bit into carry.
      outbit movcb
      \ Shift xored bit into sequence.
      [ sequence 3 + ] (#@) 2*' [ sequence 3 + ] (#!)
      [ sequence 2 + ] (#@) 2*' [ sequence 2 + ] (#!)
      [ sequence 1 + ] (#@) 2*' [ sequence 1 + ] (#!)
      [ sequence 0 + ] (#@) 2*' [ sequence 0 + ] (#!)
      drop -clue m;
```

Monitoring

In addition to hardware monitoring, you can observe the operation of **psr** with **.psr**.

The **.psr** code uses **h.**, **u.** and **d.** to display the contents of the bytes forming the 32-bit register. These display Words are very handy in displaying the results of your coding.

```
\ View current shift register
: .psr  cr
[ sequence 0 + ] #@ h.
[ sequence 1 + ] #@ h.
[ sequence 2 + ] #@ h.
[ sequence 3 + ] #@ h.
space
[ sequence 0 + ] #@ u.
[ sequence 1 + ] #@ u.
[ sequence 2 + ] #@ u.
[ sequence 3 + ] #@ u.
space
[ sequence 1 + ] #@
[ sequence 2 + ] #@ d.
;
```

```
\ Note that #@ and #! push and pop the data stack, but
\ (#@) and (#!) assume the top of stack is already free to be used, so you
\ don't need to push or pop.
```

Initialization

The **psr!** Word allows you to store a specific value in the register. Again, the use of **a** and the auto-incrementing indirect store operator simplifies the task.

```
\ Load a seed value in the shift register.
: psr! ( n1 n2 n3 n4 - )      sequence # a! !+ !+ !+ ! ;
```

The **0psr** Word just puts four zeroes on the stack, loads them into the register bytes using **psr!** and uses **?psr** to load $aa into all bytes.

```
: 0psr          0 # dup dup dup psr! ?seed ;
```

The register is initialized for application startup with **init**. It uses **0psr** to load $aa into the register bytes and then shifts the register 256 times to get a good starting value. Note that Register 7 is used for the **#for … #next** loop.

```
: init          0psr   0 # 7 #for  psr  7 #next ;
```

The Word **t** has been coded to test the operation of the algorithm interactively from the Host. It uses **.psr** to display the result of shifting the register 13 times.

```
: t     13 # 7 #for psr 7 #next .psr ;
```

Execution

The “go” Word starts the execution of the application, initializing it and then executing **psr** in an endless loop. For a Turnkey application **go** is the default startup Word. To configure the application for Turnkey operation, change the definition of **startup** in **job.fs** so that the version ending in “go” is used (it is normally commented out).

```
: go    init-xbr 0psr begin psr again
```

Finally, there are two definitions to toggle the application’s port pins so that you can verify that the hardware is working. You should be able to toggle the status LED with **~P1.6** (toggle P1.6).

```
: ~P1.6         6 .P1 toggle ;
: ~P1.7         7 .P1 toggle ;
```

Appendix A

Listings

```
\ config.fs

true constant tethered  \ Type of interpreter

\ $200 constant start  \ Reset vector.  Bootloader at $0000.
$400 constant start  \ Reset vector.  Bootloader at $0000.
\ 0 constant start  \ Reset vector.
\ $8000 constant start  \ Reset vector.  RAM at $8000.

\ $203 constant rom-start  \ Start of code, no interrupts reserved.
$403 constant rom-start  \ Start of code, no interrupts reserved.
\ $03 constant rom-start  \ Start of code, no interrupts reserved.
\ $8003 constant rom-start  \ Start of code, no interrupts reserved.

$10000 constant target-size

\ Pick downloader for your microcontroller.
: downloader  ( - )
\       s" download-ADuC.fs"
        s" download-cygnal.fs"
\       s" download-oldamr.fs"
        ;
```

```
\ job.fs (typical example)

\ --- Customize first --- /

\ ADuC841
\ 220 constant default-TH1
\ target-image rom-start 0 fill  \ nop
\ :m init-serial
\        $52 # SCON #!  default-TH1 # TH1 #!  $80 # PCON ior!
\        $20 # TMOD #!  6 .TCON set  m;

\ amrGadget
:m init-serial  m;  \ Done by the bootloader.

\ Choose an interpreter.
true value tethered
tethered [if]
         include tether.fs
[else] include standalone.fs
[then]

\ --- Then load the application --- /
include debug.fs  \ Comes before the application, useful tools.

include preamble.fs       \ 120 Boot Loader
include io.fs             \ 120 xbar
include spi120.fs         \ fast hardware SPI
include dac120.fs         \ 12-bit DAC
include main.fs           \ main application

include interactive.fs  \ Should come _after_ application for efficiency.

\ --- Finally patch the reset vector --- /

\ Turnkey or interactive.
\ start interrupt : cold stacks init-serial go ;   \ Turnkey
start interrupt : cold stacks init-serial quit ;  \ Interactive

tethered [if]  \ For interactive testing, entering numbers.
         :m # number emit-s m;
         :m ## [ dup 8 rshift $ff and swap $ff and ] # # m;
[else] headers ] here [ dict org heads ##p! org ]
[then]
```

```
\ loader.fs

0 [if]
Copyright (C) 2004-2006 by Charles Shattuck.

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For LGPL information:   http://www.gnu.org/copyleft/lesser.txt

[then]

only forth also definitions
: nowarn warnings off ; : warn warnings on ; : not 0= ;
nowarn

include ansi.fs  \ Part of Gforth.
warn
variable colors
: in-color  true colors ! ;
in-color
: b/w  false colors ! ;
: color  ( n - ) create , does> colors @ if @ >fg attr! exit then drop ;
red color >red
black color >black
blue color >blue
green color >green
cyan color >cyan
```

```
\ include vtags.fs use-tags
include tags.fs   \ part of GForth
include config.fs
include compiler.fs
include saver.fs
include dis5x.fs
downloader included

\ Forth primitives.
include misc8051.fs
rom-start org

\ This is the application.
include job.fs

report
save
[ .( Host stack= ) .s cr
```

```
\ misc8051.fs

0 [if]
Copyright (C) 2004-2006 by Charles Shattuck.

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For LGPL information:   http://www.gnu.org/copyleft/lesser.txt
[then]

nowarn

: hello ." Talk to the target " ;
' hello is bootmessage

variable talks 0 talks !
: talking  true talks ! ;

\ ----- Virtual Machine ----- /
\ Subroutine threaded.
   0 constant S \ R0 = Stack pointer.
   1 constant A \ R1 = Internal address pointer.
$e0 constant T : .T T + ; \ Acc = Top of stack.
\ DPTR = Code memory address pointer, aka P.
\ B is used by um*, u/mod, and over, not preserved.
\ misc8051.fs

\ ----- 8051 Registers ----- /
$82 constant DPL $83 constant DPH
$98 constant SCON : .SCON SCON + ;
$99 constant SBUF
$80 constant P0 : .P0 P0 + ;
$90 constant P1 : .P1 P1 + ;
$a0 constant P2 : .P2 P2 + ;
$b0 constant P3 : .P3 P3 + ;
$81 constant SP
$d0 constant PSW : .PSW PSW + ;
$88 constant TCON : .TCON TCON + ;
$89 constant TMOD
$8a constant TL0 $8b constant TL1
$8c constant TH0 $8d constant TH1
$8f constant PCON
$a8 constant IE : .IE IE + ;
$b8 constant IP : .IP IP + ;
$f0 constant B  : .B  B + ;
\ $fd constant SP0 $80 constant RP0
$100 constant SP0 $80 constant RP0

\ ----- Subroutines ----- /
\ : clean  begin key?-s while key-s drop repeat ;
: listen  begin  key-s dup 7 - while  emit  repeat  drop ;
: (talk)  ( a - ) ( clean) 0 emit-s key-s
   drop dup $ff and emit-s 8 rshift $ff and emit-s ;
\ Enabling the '[char] | emit' tags results coming from target.
\ Words executed only for the host won't do that.  A debugging aid.
: talk  ( a - ) >red ( [char] | emit) (talk) listen >black ;

:m call  ( a - )
        hint
        [ dup $f800 and ] here [ 2 + $f800 and = if
                dup 8 rshift 32 * $11 + ] , , [ exit
        then $12 ] , [ dup 8 rshift ] , , m;

:m -:  (  - )
        [ >in @ label >in !
        create ] here [ , hide
        does> @ talks @ if  talk exit  then ] call m;

:m :  (  - ) -: header m;
```

```
\ misc8051.fs

:m ;a  ( - )
        edge c@-t $1f and $11 = if
                ] here [ 2 - dup c@-t $ef and swap c!-t exit
        then ] $22 , m;

:m ;l  ( - )
        edge c@-t $12 = if
                $02 ] here [ 3 - c!-t exit
        then ] $22 , m;

:m ;  ( - )
        edge here [ 2 - = if ;a exit then ]
        edge here [ 3 - = if ;l exit then ]
        $22 , m;

\ ----- Assembler ----- /
[ \ These are 'assembler', not 'target forth'.
: interrupt ( a - ) ] here swap org dup call ; org [ ;
: push $c0 ] , , [ ;   : pop $d0 ] , , [ ;
: set $d2 ] , , [ ;    : clr $c2 ] , , [ ;  \ bit
: setc $d3 ] , [ ;     : clrc $c3 ] , [ ;   \ carry
: toggle $b2 ] , , [ ; : reti $32 ] , [ ;
: nop 0 ] , [ ;
: inc  dup 8 < if $08 + ] , [ exit then $05 ] , , [ ;  \ Rn or direct
: dec  dup 8 < if $18 + ] , [ exit then $15 ] , , [ ;
: add  dup 8 < if $28 + ] , [ exit then $25 ] , , [ ;
: addc dup 8 < if $38 + ] , [ exit then $35 ] , , [ ;
: xch  dup 8 < if $c8 + ] , [ exit then $c5 ] , , [ ;
: ##p! $90 ] , [ dup 8 rshift ] , , [ ;
\ : mov  $85 ] , swap , , [ ;
: mov dup 8 < if $a8 + ] , , [ exit then
        over 8 < if swap $88 + ] , , [ exit then
        $85 ] , [ swap ] , , [ ;
: movbc $a2 ] , , [ ;  \ Move bit to carry.
: movcb $92 ] , , [ ;  \ Move carry to bit.
: [swap]  $c4 ] , [ ;  \ swap nibbles
```

```
\ misc8051.fs

\ ----- Conditionals ----- /
:m then hide here [ over - 1 - swap ] c!-t m;
:m cond hide , here 0 , m;
:m if   $60 cond m;  :m 0=if $70 cond m;
:m if' $50 cond m;   :m 0=if' $40 cond m;
:m if. $30 , cond m; :m 0=if. $20 , cond m;
:m -if 7 .T if. m;   :m +if 7 .T 0=if. m;
:m begin here hide m;
:m end [ dup >r 1 + - r> c!-t ] hide m;
:m until if end m;
:m 0=until 0=if end m;
:m until. if. end m;
:m 0=until. 0=if. end m;
:m -until  -if end m;
:m again call ; m;

\ ----- Stack operations ----- /
:m nip [ S inc ] m;
:m drop hint $e6 , nip m;
:m dup S dec $f6 , m;
:m swap $c6 , m;
:m (over) $86 , B , dup $e5 , B , m;
:m 2drop nip drop m;

\ ----- Optimizing ----- /
\ The hint helps #, doesn't hurt anything else?
:m ?dup  (  - ?)
        edge here [ 2 - - if ] hint dup [ exit then
        edge @-t $e608 = if
                -2 ] allot here [ there 2 erase exit
        then ] hint dup m;

:m ?lit  (  - ?)
        edge here [ 4 - - if 0 exit then
        edge @-t $18f6 = ] edge [ 2 + c@-t $74 = and if
                ] here [ 1 - c@-t -4 ] allot here
                [ there 4 erase -1 exit
        then 0 ] m;

:m =if ?lit [ 0= if abort then ] $b4 , cond m; \ Does literal = T?.
:m <if =if then if' m; \ Is T <= literal?.
:m =until  =if end m;
:m <until  <if end m;
```

```
\ misc805

\ ----- More stack operations ----- /
:m (#)  $74 , , m;     :m # ?dup $74 , , m;
:m ##   [ dup ] # [ 8 rshift ] # m;
:m ~#  [ invert ] # m;
:m push [ T push ] drop m;        :m pop ?dup [ T pop ] m;
:m SP! $75 , S , , m;             :m RP! $75 , SP , , m;
:m stacks SP0 SP! RP0 RP! m;

\ ----- Arithmetic and logic ----- /
:m 1+ $04 , m;        :m 1- $14 , m;
:m 1u+ $06 , m;       :m 1u- $16 , m;
:m invert $f4 , m;    :m negate invert 1+ m;

:m logic ( opcode) [ >r ] ?lit [ if r> ] , , exit [ then r> ] 2 + , nip m;
:m +   $24 logic m;
:m +'  $34 logic m;
:m ior $44 logic m;
:m and $54 logic m;
:m xor $64 logic m;

\ Don't use # after the SFR, a special case.
:m logic! ( opcode) [ >r ] ?lit [ if ]
  [ r> ] , [ swap ] , , [ exit then r> 1 - ] , , drop m;
:m ior! $43 logic! m;
:m and! $53 logic! m;
:m xor! $63 logic! m;

:m (u/mod) swap $86 , B , $84 , $a6 , B , m;
:m (um*) $86 , B , $a4 , swap $e5 , B , m;
:m (*) ?lit [ if ] $75 , B , , $a4 , [ exit then ]
  $86 , B , nip $a4 , m;
:m 2*' $33 , m;  :m 2* clrc 2*' m;
:m 2/' $13 , m;  :m 2/ [ 7 .T movbc ] 2/' m;
```

```
\ misc8051.fs

\ ----- Memory access ----- /
:m (#!)  [ dup 8 < if $f8 + ] , [ exit then ] $f5 , , m; \ No drop.
:m #!  ?lit [ if
          over 8 < if swap $78 + ] , , [ exit then ]
          $75 , [ swap ] , , [ exit
     then ]  (#!) drop m;

:m (#@) [ dup 8 < if $e8 + ] , [ exit then ] $e5 , , m;  \ No dup.
:m #@ ?dup (#@) m;

:m a ?dup $e9 , m;
\ Use of A is not reentrant, push and pop where needed.
:m a! ?lit [ if ] $79 , , exit [ then ] $f9 , drop m;
:m @ ?dup $e7 , m;
:m @+ @ $09 , m;
:m ! $f7 , drop m;
:m !+ ! $09 , m;

:m #for  ( direct - ) #! begin m;
:m #next  ( direct - ) [ dup 8 < if ] $d8 or cond end exit [ then ]
     $d5 , cond end m;

:m |p  ?dup $e5 , DPL , dup $e5 , DPH , m;
:m |@p  dup $e4 , $93 , m;
:m p!  $f5 , DPH , drop $f5 , DPL , drop m;
:m p+  $a3 , m;
:m |@p+  |@p p+ m;
:m (!x)  $f0 , m;
:m !x  (!x) drop m;
:m !x+  !x p+ m;
:m @x  ?dup $e0 , m;
:m @x+  @x p+ m;

0 org : reset

:m see ' >body [ @ ] decode m;
```

Appendix B

Commands & Files

Command	Description
c	Execute from a Command Prompt -- Compiles an application contained in **job.fs**. Produces **chip.bin** and **chip.hex** image files
d	Execute from a Command Prompt -- Downloads a compiled application to the Target processor.
decode <addr>	Execute from a MyForth prompt -- Decompiles starting at the specified address. Hex numbers must start with a “$” (or “\$” for Linux users).
see <word>	Execute from a MyForth prompt – Decompiles the specified Word, line by line. Use the space bar or any key but escape to display the next line. Use **q** or **Esc** to stop the listing.
sees <n> <word>	Execute from a Command Prompt after compiling your application with **c** or **d** – Decompile n lines of specified Word.
n	This is an alias for next. When used in the context of the **see** or **decode** decompilers, it will display the next line of decompilation. Any key except **q** and **Esc** will also display the next line.

File Name	Description
config.fs	Contains configuration information for your application. You must manually edit this.
job.fs	Contains the definitions and files to be included to make up your application. This is the file compiled when you execute the **c** or **d** commands.
main.fs	By convention, this is the main application file included by **job.fs**. This file is optional. It is commonly used to contain the bulk of your application. You can load your application using **job.fs** without a reference to **main.fs**.
chip.bin	Contains the compiled image of your application
chip.hex	Contains an Intel Hex representation of **chip.bin**
tags.log	Contains tag names for Forth definitions and their defining files that can be used with editors such as Vim or EMACS to go to definitions from the text editor (e.g., placing the cursor on the word and pressing “CTL-]” in Vim). For use with VIM (highly recommended), a reference to this file must be put in the Vim program directory, as described in the Editor section.
ansi.fs	ANSI terminal Word definitions for GForth (used for coloring, etc.)

Appendix C

Vim Basics

1. For Windows users, the normal cut, copy and paste shortcuts apply:

Ctrl-C	Copy highlighted text
Ctrl-V	Paste copied or deleted text
Ctrl-X	Cut copied text
Ctrl-A	Highlight entire document
Shift-End	Highlight text from cursor to end of line

The Page Up, Page Down, Delete, Insert, Home and End work the same as in Windows.

2. You can use the GVim pulldown menus to perform editing or to see the equivalent GVim commands for various menu options (e.g., :w for save on the File menu)

3. There are three GVim modes (takes some getting used to):

INSERT use this to insert text. It is invoked with the "i" or "a" commands. It is also invoked automatically -- check to see if you are in insert mode by looking for "-- INSERT --" at the bottom left of the display. **TO GET OUT OF THE INSERT MODE, PRESS ESCAPE!**

EDIT This is the mode when the cursor is not at the bottom of the display or when "-- INSERT --" is not displayed at the bottom left of the display. In the edit mode, you can use editing and navigation commands such as "10gg" to go to line 10.

COMMAND You invoke command mode by typing ":" in the EDIT mode. When you are in the command mode, the cursor will be at the bottom left of the screen and the line will begin with a colon. **TO GET OUT OF THE COMMAND MODE, PRESS ESCAPE.**

================ SPECIAL CHARACTERS, ETC. ===================

$	last line in file, end of current line
1,$	range from first line to last line
%	all lines in file (same as 1,$)

===================== DISPLAY ============================

:split <file>	split window, specified file in new window
:split .	split window, show file/directory tree
:split	split window, Windows file selector

========================= EDITING ==============================

dd	delete [count] lines into register x (dd - del. current line)
ESC	return to normal mode
gf	goto file under cursor
a	insert text after cursor (starts insert mode)
i	insert text before cursor (starts insert mode)
p	insert (place) text from buffer (use with y to cut/paste)
.	repeat last command, show file/directory list
u	undo last change
x	delete character under cursor
yy	(yank) copy current line to buffer (use with p to cut/paste)

======================= COMMANDS ============================

:e	open file
:w	save current file
:wqa	save and exit
:qa!	quit without saving
:sav	save as
:s	substitute (e.g., :1,200s/hello/goodbye or :%s/hi/bye)
:! <cmd>	execute the specified shell command (e.g., ":! dir")
ZZ	write current file to disk and exit (caps are important)

====================== NAVIGATION ============================

<line>gg	goto line
w	move forword word
/<str>	search forward for specified string
?<str>	search backward for specified string

Note: use "\" before special characters in the search string.

====================== OPTIONS ==============================

:set <option>

nu -OR- number	show line numbers
nonu	no line numbers
autoindent	indent to match previous line
ignorecase	ignore case on searches
shiftwidth=width	width of columns when using autoindent
tabstop=spaces	tab stop size
wrapscan	search from beginnig of file when end is reached

===================== EXAMPLES ==============================

:1,$s/x/y/g	substitute y for every first instance of x in lines 1 to last
:s/x/y	substitute y for first instance of x in current line
:%s/x/y/gc	substitute y for x global, with confirm