MyForth

REFERENCE MANUAL

Revision 8.4
January 24, 2011

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

Document: MyForth_Reference_Manual.doc

Contents

Contents

Contents

Contents

Contents

Contents

Contents

1

Introduction

Purpose

This manual provides general and technical information for MyForth, an 8-bit Forth for 8051 family processors written in GForth. Because MyForth is hosted by GForth, it 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 AM Research's amrForth provides a very mature, robust and full-featured 16-bit Forth for microprocessor development, Charley designed MyForth to explore and apply several of his ideas about 8051development that were not feasible within the context of amrForth.

One reason for the departure from amrForth is that it uses a 16-bit Forth model: Charley is convinced that an 8-bit model is more appropriate for an 8-bit machine. Although 16, 32 and larger bit operations are sometimes needed for tasks such as scaling, these can be considered as special cases to be coded as needed. Mostly, 8051 programming deals with 8-bit numbers and operations.

Another reason that MyForth has been developed separately is that many of its features are implemented in a "non-standard" way. This may be of concern to some, but the intent is more to explore new territory than to produce a commercial product.

Introduction

The objectives of MyForth are to:

- Implement a development environment specifically designed for 8051 family processors
- Retain the advantages of a Forth development environment while improving the performance of the compiled code
- Simplify the development environment and the underlying system code so that it can be easily used, understood and changed
- Provide a simple user interface that can be easily learned and used in both Windows and Linux environments

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

The result is a high performance Forth in a very small package that provides all of the tools needed 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 a Forth programmer who wants to use MyForth to develop applications but needs to know how to get started. The manual also provides reference information for more experienced users and for those who want to understand more about how the system works.

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

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 broader audience. Like MyForth, 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 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. However, Linux users should find that most of the information in this manual is directly applicable to a Linux 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 (SL) development systems. These provide the EC2 serial adapter or USB Debug Adapter and the software needed to load the MyForth bootloader with the SL Integrated Development Environment (IDE).

After the bootloader is installed, the USB Debug Adapter and its software are no longer needed: all MyForth programming is thereafter performed via the serial port.

If you already have a debug adapter, you can simply buy a Silicon Laboratories Target Board and use it. These boards are very inexpensive, some selling for $50 or less. The Target Boards for “top of the line” chips such as the C8051F120, are available for approximately $100.

Another excellent development option is to use Silicon Laboratories’ Toolstick line of development products. These can be programmed with the standard Silicon Laboratories IDE and a USB programming adapter that costs under $20.

The Toolstick daughter (Target processor) cards are very compact (about 1.5” x 1 3/4”) and provide a processor and a surprising amount of support circuitry. For example, the Toolstick for the C8051F362 processor contains a 100 MHz processor with three 8-bit I/O ports. The Toolstick I/O is available on 0.1” spaced pads and on-board circuitry includes a voltage regulator, SMD power and status LEDs, and a test pushbutton. All of this costs about $10 in single quantities and is available from vendors such as Mouser and Digikey.

MyForth can also be used with the AM Research Gadget Development System (no longer sold commercially). The MyForth serial bootloader is compatible with the AM Research Boot Loader that is loaded on every Gadget board.

Scope

It is assumed that the user is already somewhat familiar with the 8051 instruction set, assembly language programming and the basics of 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:

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

The following are outside the scope of this manual:

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

Conventions

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

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” denotes the Target processor. For example, the Target processor for a C8051F120-TB Target Board (TB) from Silicon Laboratories (SL) 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-based system “Hosting” GForth and the MyForth system. It consists of the PC hardware and various programs and facilities working together to provide a development environment. Most references to the Host are generally references to GForth or the MyForth system implemented with it.

The programs and facilities residing on the Host include the MyForth system files, GForth, an editor and the operating system.

The Host compiles MyForth statements into 8051 assembly code stored in a Target image residing in the GForth memory space. This image can be examined and downloaded to the Target via a serial port and the Boot Loader. The image is also stored in image files that can be downloaded to the Target, either by the MyForth downloader or by the Silicon Laboratories IDE and USB Debug Adapter, EC2 Serial Adapter or Toolstick programmer.

Tethering

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

The Target code is minimal because it relies on the Host to provide most of the user features.

To maintain the tether, the Target runs a program that executes a single command. When this command is executed, the Target provides a status feedback to the Host. The command executed by the Target receives two address bytes from the Host and then executes code at that address. When the code at the address has been executed, the Target sends a “done” byte back to the Host to terminate the exchange. The Host’s 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 Target addresses.

Tethering provides convenient user interaction while minimizing overhead on the Target (e.g., no Target-based dictionary or complex interpreter are required). Tethering is explained in more detail in a later section.

Turnkey

If the Target application needs to execute autonomously without user interaction, the Target image can be configured as a “Turnkey” system that starts up executing user code in a continuous loop. This mode is commonly used when the final application is deployed.

The turnkey, standalone and tethered (interactive) options are set up in a job control file which also controls the loading of Target source and Target configuration.

Standalone

The Target can also be configured as a standalone Forth system by compiling a standalone system. The standalone system is most often configured as a turnkeyed application that interacts with a user or remote processor over a serial link.

Words

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

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

Normally, 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 (turnkey operation).

Code Words

The function of Code Words is the same as Words. They can be executed from the Forth command prompt. But, unlike Colon definitions, Code Words are defined 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 Words that compile assembly language instructions into the Target image (residing on the Host PC) when they are executed. MyForth handles macros by changing search orders. Macros are defined using the **:m <name> … m;** sequence. Later sections explain this in more detail.

The important thing to know about macros is that, when executed, they assemble instruction sequences. The instruction sequences assembled by Macros cannot be executed standalone at a MyForth prompt. Because of this, they must be included inside a MyForth Word and executed as a Forth Word (e.g., from a command line).

In Line Assembly

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

2

Installation

Overview

To use MyForth you must first install GForth, a mature, free, LPGL Forth that is commonly available on the Internet. Installation is straightforward. MyForth assumes that Gforth is installed to the default directories.

It is also recommended that you install the Vim (GVim) editor, also widely available on the Internet, as described below in the Editing section.

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

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.

You can get a copy of MyForth by sending an email request to Bob Nash at Bob.Nash1@gmail.com .

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

Windows Development

Command Prompt

Because MyForth was developed in a Linux environment, it is designed to execute from a command line. Windows compatibility must be 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 to be entered at the Command Prompt and the typical development session will mostly occur within the context of the GVim editor or the MyForth 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.

You can find the Command Prompt program in the Start/Accessories menu. It is labeled “Command Prompt” and its icon looks like a Window with a black background with a white “C:\” prompt on it. We suggest that you put it on the desktop and change the default startup directory.

To change the default startup directory, first right click on the desktop icon and select the “Properties” menu option. Next, edit the “Start In” path to be “C:\MyForth\Projects” so that you will always start up ready to navigate to one of your projects. You may also want to create a batch file, as described below, to easily navigate to a particular project.

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, right click on the Command Prompt icon and select the “Properties” option. When the Properties dialog opens, select the Colors tab and change the background to white and the text to black or blue.

Startup

To start up in one of your project directories ready to start development, you can create a batch file that automatically takes you to a particular project directory.

If you have changed the Command Prompt to start up in the Projects directory, we recommend putting the batch file in the **C:\MyForth\projects** directory. Here is an example of a batch file, **psr.bat**, that will take you to the psr Project directory and display the files therein in a compact format:

```
cd \MyForth\projects\psr
dir /w
```

A more elegant way to automatically start up in your project directory is to create a batch file that takes the name of your project and changes to its project directory with “switch” statements. This “switcher” batch file should be put in your path statement.

Here is an example of typical statements from a “switcher” batch file:

```
SET LOCAL=c:\myforth\projects
SET DEVDRIVE=c:

:: ----- LOCAL DEVELOPMENT -----

if "%1" == "eput" (
  %DEVDRIVE%
  cd %LOCAL%\eput
  GOTO:EOF
)
if "%1" == "300-dev" (
  %DEVDRIVE%
  CD %LOCAL%\300-dev
  GOTO:EOF
)
:END
```

Application Development

Development commands consist of four batch files that are contained in a local application development directory. There are only four batch commands: **c**, **d**, **run** and **sees**.

The **c**, **d** and **sees** batch commands are used to compile, download and decompile code, respectively. During routine application development you will mostly use the **c** and **d** commands.

The **run** command allows you to run MyForth and GForth commands in a script. This is quite useful when producing formatted reports to document testing.

The **sees** batch allows you to decompile your application; it is rarely used and is primarily invoked with command line redirection to generate a listing file to document the decompilation of a section of code.

For Windows users, the above batch 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.

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 both much simpler to use and to 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 GVim editor (the Vim editor using a GUI). GVim is a free Vi-compatible editor that allows you to edit text in both GUI and command line modes. It also performs operations such as shelling out for command execution and editing definitions from multiple files. GVim can also be configured to perform color highlighting based on language syntax. This feature is used by MyForth to help interpret both source code and decompiled code.

Besides being free, GVim is also available for both Windows and Linux and works the same in both environments.

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 highlighting is easily implemented with GVim but may be more difficult with other editors.

GVim is widely available on the Internet. In Windows, install it in the default directory (**C:\Program Files**). Afterward, install, replace or edit a few configuration files, as described below. Except for the change to ***autoexec.bat***, all necessary configuration files are supplied with MyForth.

Usage

If you have never used Vim, GVim 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.

GVim 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.

One of the more useful features of GVim is its ability to easily navigate between files by placing the cursor on the file name and executing the **gf** (go file) command.

Appendix B contains a summary of commonly used GVim commands.

Note that this manual tends to use the terms “Vim” and “GVim” somewhat interchangeably. This is because there are two executables furnished with the Vim installation, ***Vim.exe*** and ***GVim.exe***. You can use either but will probably prefer to use GVim, which is the “GUI” version of Vim. Also, Vim lacks some of the features discussed in this manual.

Path

Because development with GVim requires executing it from a command line or using batch file shortcuts, you should edit the Windows path to include Vim, GForth and Cygwin (if used). On Windows XP, you can edit the path by selecting the Control Panel on the Start menu and then selecting “System.” In the System panel, select the “Advanced” tab and click on the button labeled “environment variables.” There you can select and edit the “path” environment variable.

With this addition, you can now edit MyForth files using the “e.bat” shortcut (e.g., **e job.fs**). Note that you may need to reboot Windows for the **path** command to become effective.

Tags

An important feature of GVim when used with MyForth is that it can use the **tags** file, **tags**, that GForth automatically updates whenever a Word (or header) is defined. Using this feature, you can switch to the file defining a Word (or macro) by putting the cursor on Word and pressing **Ctrl-]** (go back with **Ctrl-T**).

You can also request a search for a specific tag at the GVim command line. For example, to search for the Word “init”, enter: **:tag init**.

To use **tags** with Vim and GForth, you may need to change the default GVim settings. The default behavior of GVim is to assume that the **tags** file is sorted. However, GForth writes the **tags** file as it compiles definitions. This causes GVim to issue a “tags file not sorted” error message.

To fix this problem, edit the **_gvimrc** file, adding this line: **set notagbsearch**. This line is included in the sample **_gvimrc** file supplied with MyForth.

Note that GForth automatically generates a **tags** file in each MyForth project directory whenever you execute a **c** (compile) or **d** (download) command.

Navigation

Vim allows you to navigate easily between files and the definitions in them. We suggest starting your editing sessions with **gvim job.fs** even if you know the name of the file or Word that you want to edit.

For convenience, the file **j.bat** is included in the MyForth files that are copied to a new project. To use it to start up editing your project's Job file, go to your Project directory and enter the **j** command at the command prompt. Note: there is no **j** command file for Linux.

From the Job file, **job.fs**, you can easily navigate to just about any file in your application. To do this, assuming you are in command mode, put the GVim block cursor on the name of the file to be edited and enter the **gf** (go file) command. If you have navigated to a file this way and want to go back, press **Ctrl-6**.

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-]**. The editor will go to the file and position the cursor near the Word's definition.

Another useful GVim 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**. To simplify the process of editing a tagged word, the MyForth system files include the **t.bat** file.

To use the **t** command (batch), execute it followed by the Word you want to edit. For example: **t emit**. GVim will search the **tags** file for the name of the file containing the definition of "emit" and open the file with the cursor just below it. Of course you must have compiled your application earlier so that the Word appears in the tags list.

To recompile your application and refresh the tags file, transfer to your project's Project directory and enter the **c** command at a Windows Command Prompt. After recompiling, enter "bye" at the MyForth command prompt and use **t <word>** to edit your Word. Note that the **d** command also recompiles your application and refreshes the **tags** file, but this command normally precedes a download, not an editing session.

Colors

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.

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 languages, but how to do that is not covered in this manual.

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.

All of the syntax highlighting files can be found in the **MyForth\vim** directory. This directory also contains a **VimNotes.txt** file that covers the installation and function of the files in the **vim** directory.

3

Projects

Overview

The following describes how to start 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 system files to it and editing configuration information in the Job file.

The process of compiling and downloading a new project is quite straightforward and is explained in the Job Control section below.

Project organization is a matter of taste and the following sections describe the way that the author normally organizes small projects. For larger projects, a different organization may be required. The following discusses several project initiation and development approaches and the rationale behind the organization recommended by the author.

For new users, using the recommended organization provides a simple way of getting started with the examples and projects given in this manual.

New Project

Project Directories

MyForth projects are contained in project directories. These directories are normally created in subdirectories under the **MyForth\projects** directory.

For example, a new project named “myproject” would reside in the **MyForth\projects\myproject** directory.

System Configuration

Once you have created a new project directory, copy all of the system files from the **\MyForth\system** directory into it. This establishes the system files needed for a new project. Some of these files may have to be edited before you can interact with your Target processor, as described later.

After copying the system files into your project directory, you have all of the system files you will need for development, but your project will not be configured for a particular Target processor.

Processor Configuration

To configure for a particular Target processor, you will need three processor specific files:

1. The bootloader file
2. The Special Function Register (SFR) file
3. The job template file

Copy these three files from the **\MyForth\chip** directory to your project directory. For example, if you want to create a project for the Silicon Laboratories C8051F300 chip, copy the files **bootloader300.fs**, **sfr300.fs**, and **job300.fs** from the chip directory to your project directory.

Quick Configuration

A quick project configuration option is to copy the files from an existing project that uses a configuration that is similar to the one you will be using for your new project.

This is somewhat easier than copying files from two separate directories, but does not guarantee that you will be using the latest distribution versions of the system and configuration files.

If you copy files from a previous project directory, they may not contain the versions from the most current release or may contain files that have been modified for a particular application. Perform a quick configuration only if you know that the template application was developed with the current release files and with compatible modifications.

After copying files from the system and chip directories or from a previous project, you can develop using just the files in your project directory; development will be independent of the files in the **system** and **chip** configuration directories.

Rationale

The two configuration procedures described above ensure that when changes are made to the **system** and **chip** 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 the MyForth system or processor (chip) files.

Copying the system files to a project directory for each new project may seem to be wasteful but this is not much of a problem because of the small size of a complete development image (about 200K and 35 files).

Note that both Linux and Windows command and serial configuration files are copied to each new project directory. This allows you to easily change environments later.

Using a single central directory for system files would save memory and ensure that all projects use the latest revisions of the system files. Version changes in system files can be handled with a backup or version control system, per normal practice. However, the total size of MyForth is small enough that using a local system directory does not require much additional memory.

For larger projects a central system directory and a version control system (such as Subversion) are recommended. This method has been used for one large MyForth project and works well. But, for small single-processor projects that can be contained in one directory, the “atomic” method described above is recommended.

One final note: MyForth is relatively mature and there have been only a few minor changes to system files over the past few years. There is little risk that a project will not operate properly if system files are compiled from the **system** and **chip** directiories.

The major configuration risk is in the **chip** files. The most common change is in the bootloader to accommodate a different baud rate (several alternatives are commonly shown but commented out. Also, the SFR files are periodically upgraded; the current versions do not define all SFRs and may need to be upgraded for a new project that uses new chip capabilities.

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 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 the Silicon Laboratories C8051F300 Target Board, you can interactively compile and 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, you may want to put the **gg.bat** file from the system directory in a directory that is included in the "path" environment variable. This batch file is an "application switcher" that was mentioned above. With this file in the path, you can change to the examples directory by entering "gg examples" any command prompt.

Whenever you create a new project directory, you may want to edit the **gg.bat** file to include the new project. This way, you can navigate to the project directory with the "gg" command.

Job Control

The compilation of a MyForth application is controlled by the **c** and **d** commands and the job control file, **job.fs**.

The **c** and **d** commands are very similar in structure. The **c** command compiles the application and brings up a MyForth interactive session with the Target. Like the **c** command, the **d** command compiles the application, but it also downloads it to the Target before bringing up an interactive session.

Both the **c** and **d** commands load the Job file to compile the MyForth application.

To understand how a new application is compiled and downloaded, it is useful to understand how these three components work. The following sections describe their operation in more detail.

Command Files

The **c** and **d** command files are implemented with the **c** and **d** scripts on Linux systems and by the **c.bat** and **d.bat** command files on Windows.

Examining the **c** and **d** command files reveals a long and complicated-looking command line. This line, although it looks intimidating, can be quite easily understood by looking at the individual tasks it performs:

1. A comments section provides a revision history and an overview of path ordering and how to change the file for different com ports and baud rates.

2. The command line invokes GForth to perform all of the job control tasks, including setting the GForth path variable, so that system source code files can be found.

3. Com port and baud rate variables are established before calling GForth's serial communications facilities to set up an interactive serial link with the Target.

4. The Job file loads to compile the application.

5. After compilation, the pre-configured com port is opened, the compiled code is downloaded to the Target (**d** command only), and the tethered interactive session is established with the Target. If the commands execute correctly you should see the MyForth prompt.

Comments Section

The comments section provides information on how to change the **c** and **d** command files for different com ports and baud rates.

Often, applications use different com ports and baud rates. Putting the command files is the root project directory ensures compatibility with the communication rates used by the project (e.g., as established in the bootloader source file. Note that the bootloader establishes the baud rate. If there is a configuration problem check that the **c** and **d** command files are using the same baud rate established in the bootloader.

By executing the **c** and **d** commands from the project directory, you are assured that the baud rate is compatible with the application's serial settings established in the bootloader.

Source Path

Following the comments section in the **c** and **d** command files, the command line invokes GForth, using the "-e" option to execute GForth code contained in the following parentheses. The **fpath** command is used to establish the GForth search path set by the "path=" statement. This statement specifies a number of paths, in path search order, separated by vertical bars.

For example, the statement "fpath path= ./|./system|C:\PROGRAM~1\gforth" would specify that GForth should first search the local project directory (./) followed by a local system directory and, finally, the GForth directory.

Once the source code search path is established, GForth executes **.fpath** to display the path when the application is compiled.

COM Port Settings

Following the display of the search path, another "-e" option is invoked to set the com port and baud rate variables, **com?** and **current-baudrate**. Values for these are given in the comment lines of the command files.

The com variables are used when GForth executes the statements included in the **serial-windows.fs** or **serial-linux.fs** source files. The serial configuration file is loaded by simply including it on the GForth command line. When loaded, this file establishes the serial port commands to be invoked later.

Job File

The next item in the GForth command line is "./job.fs" which specifies that the Job control file, **job.fs**, should be loaded from the current (project) directory. The Job file does the heavy lifting in the compilation process, as described in a following section.

Note that, when copying processor-specific files from the **chip** directory, the job file contains the name of the processor (e.g., job300.fs). This is necessary to distinguish between different processor configuration templates in the job file.

Before using the command files, the job file name must be edited to be "job.fs."

Target Communication

The last sequence executed in the **d** command file is "open-comm download target talking" which tells GForth to open the comm. port, download the executable image to the Target, and establish communication with it.

The **c** command file is identical except that it does not execute the "download" command to download to the Target.

Thus, the "c" command can be used to re-establish communication with a program running on a tethered Target. This command can also be used to perform a test compile, and does not require an operating Target. Typical use would be to check for compilation errors or to examine compiled code with the "see" or "decode" commands.

Job Control

The Job file controls the configuration and compilation of each application. The following sections describe the items that you will likely find in the Job file and their functions. See the Job file for one of the distributed applications for a better understanding of what a typical Job file looks like.

An noted above, job file templates for various processors are contained in the **chip** directory. For a new project, the template file, such as job300.fs, is copied to the new project directory. ***Before first use, this file must be renamed to "job.fs."***

Using the Job file to include all major functional components is a convenient way to configure your application, to load it in the proper sequence and to organize it.

A convenient way to navigate to all parts of an application is to start editing sessions by executing **gvim job.fs** or by executing the shortcut command files, **j** or **j.bat** to perform the same command sequence.

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 also back out through the editing file chain by entering **Ctl-6**.

Processor Configuration

The first items at the beginning of a Job file are definitions for constants that determine the interpreter type, turnkey option and the compilation of local headers.

Typically these would include a definition such as “true constant tethered” to specify that a tethered interpreter should be used instead of a standalone interpreter.

The first processor-specific definition is for “start”, the cold start reset vector. This is used later in the Job file to patch the reset vector. It is discussed in more detail later in this section.

Following the definition for “start” are allocations for the remapped interrupt vectors. MyForth application code starts at the location specified by “rom-start”, which is always just after the last-allocated (relocated) interrupt vector.

For example, if the last interrupt vector starts at $20B (timer 0), then “rom-start” could start at $20D, two bytes after it (to allow for the jump vector code).

If you have interrupts vectors other than the Cold Start vector (start), you must change the ROM start location to allow for them. This manual re-allocation of interrupt vectors could be automated, but the price would be added complexity and wasted memory resources.

The MyForth rationale for manually changing the start of your code to be after the last interrupt vector 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 the current interrupt configuration is not particularly onerous.

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

The last processor-specific configuration value is "target-size." It specifies the amount of flash contained in a chip so that the compiler knows how much memory to allot for a code image.

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.

To properly configure your project, it is useful to examine the Job files for projects that have previously used the same processor. The Job files for these projects can be used to provide a starting point for your project's configuration.

For projects using the same processor, configuration generally consists of editing the interrupt and "rom-start" locations and choosing the appropriate interpreter.

Note that the largest configuration difference you will find will be between the C8051F120 processor and other smaller processors such as the C8051F300, 310 and 410. This is because the page size for the 120 chip is larger and the reset vector must be remapped to $400 instead of $200 for the other smaller iprocessors.

System Compilation

After the configuration statements, the Job file includes **loader.fs** (the Loader file or Loader). The Loader compiles the MyForth system using the configuration information previously set in the Job file.

The Loader first sets up terminal color options and then "includes" all of the files needed to build MyForth. The following is a list of the files included by the Loader with some comments describing what they do:

```
include gforth/vtags.fs use-tags
                                \ part of GForth: tags file
include ./compiler.fs           \ load the MyForth compiler
include ./saver.fs              \ code to write chip.bin and chip.hex
include ./dis5x.fs              \ the 8051 disassembler
include ./download-cygnal.fs
                                \ downloader definitions
include gforth/dumb.fs          \ command line dumb terminal,
                                \ useful when talking to an
                                \ application with a standalone
                                \ interpreter

\ Forth primitives.
include ./misc8051.fs           \ MyForth 8051 compiler
rom-start org                   \ needed by following code
```

Some of these files are discussed more detail in later sections.

Bootloader Image

Except for special cases, code images compiled by MyForth include a Bootloader. Bootloader code is normally included in every programming image (e.g., the **chip.hex** and **chip.bin** files). This allows this file to be used with the Silicon Laboratories debug adapter and IDE to initially program a chip that does not have a MyForth Bootloader.

The Bootloader image is compiled by including the bootloader for the chip used in your project. For example, if the project is for a C8051F410 chip, then the Job file would load **bootloader410.fs** . Before loading the bootloader, definitions for the special function registers are needed. These are defined by loading them in the job file with a statement such as: **include ./sfr410.fs** .

A “rom-start” statement follows the compilation of the Bootloader to ensure that following code is compiled at the start of program code, just after the remapped interrupt vectors, as set by earlier statements in the Job file.

The bootloader code is compiled just after the last possible re-vectored interrupt in low memory. For example, if the last remapped interrupt vector is at $083, then the bootloader begins at $08B. The bootloader size is approximately 250 bytes.

You can find the start of the bootloader code by examining the bootloader file for your processor and looking for the comment that says something like “Code starts here, after interrupts.

You can also find the location of the cold start vector (the Word “start”) by examining the contents of location $0000 (e.g., using the SL IDE) and observing the jump location. A typical jump location would be to $00BD.

Interpreter

Before loading the application, you should choose an interpreter. The Job file includes code that looks like this:

```
\
\ -----[ Choose an interpreter ]
tethered      [if] include ./system/tether.fs [then]
standalone [if] include ./system/standalone.fs [then]
```

If the “tethered” constant is set to “true” at the start of the job file, then a tethered interpreter will be loaded; if “tethered” is set to “false” and if “standalone” is set to true, then the standalone interpreter will be loaded.

Note that you can have both a standalone and a tethered interpreter. Typically this is done during development of an application that will use a standalone interpreter but requires interactive debugging during development.

Application

At this point in the Job file, the base MyForth system has been installed, complete with a bootloader and an interpreter. Your application can now be compiled by using “include” statements to load your application modules.

To build a complex application, you will “include” several files, using an “include <filename.fs>**”** command sequence for each file “included” in the Job file. These include files load your application’s source code files, including library files, device drivers, utilities and application source code.

Normally, you will want to include the file **debug.fs** at the start of your application to provide useful debug tools. These include utilities such as “h.”, “dh.”, “u.”, and “ud.” to display numbers from the Target’s stack in a convenient format (e.g., as hex or double numbers).

Your application’s source code modules follow the debug definitions. After the project is complete, you can comment out the debug definitions to save memory.

It is suggested that every application include an initialization file (e.g., **init.fs**) that is loaded after all other application code is compiled. This file initializes your entire application and typically provides the following:

1. A comment section containing a complete list of all chip I/O assignments and types. This list should be updated as application code is added that consumes processor resources or requires special configuration (e.g., analog inputs).

2. A master initialization Word that sets up the crossbar, sets the port I/O configuration, etc. This Word should also include initialization Words for all application resources. A “standard” used for many MyForth applications is to name the master initialization Word “/chip” or “init”. Words that initialize hardware for a particular device should be defined in the module for that device and executed as part of the master initialization Word (e.g., /lcd for lcd I/O).

It is also suggested that every Job file contain a **main.fs** file that defines the main application loop and the turnkey Word, **go**.

After the Main definitions, most MyForth applications include a file named **interactive.fs** . This file contains a number of definitions that are useful when exercising applications interactively. It can be commented out after the application is complete.

Reset Vector

The processor's normal reset vector at $0000 is patched to jump to MyForth's boot loader. Thus, when the processor is reset (e.g., via the reset switch), the bootloader is the first to execute. It checks for download requests and, if there are none, it times out and jumps either to the tethering code (to establish a tethered session) or to the start of the turnkeyed application (i.e., the "go" Word).

To patch the processor reset vector, it is necessary to remap all of the other 8051 interrupt vectors. At each of the old interrupt vector locations in low memory MyForth installs a jump to an interrupt vector located elsewhere in low memory.

The remapped vectors can reside at several different locations, depending on the Target processor. For most Silicon Laboratories chips, the remapped reset vectors will start at location $200, as discussed earlier. Because of the larger page size for the C8051F12x series processors, the remapped reset vector location is $400.

The reset vector, also called the cold start vector, is patched after your application is loaded. If your application is turnkeyed with a "go" Word, then the cold start vector is patched with the following statement:

start interrupt : cold stacks go ;

The "interrupt" Word takes the "start" location (set earlier in the Job file) and installs a jump to the Word "cold", which is defined following "interrupt."

In the example above, cold clears the Target stacks and executes the turnkey Word, "go", which is usually defined in **main.fs** . The "go" Word performs application initialization (e.g., with **/chip**) and then executes the main application loop.

Dictionary

For projects compiled with a Standalone Interpreter, a local dictionary is compiled and downloaded. So that the Standalone Interpreter can access the dictionary headers, the start of headers location must be patched into the dictionary pointer, **dict**, which is defined in the Standalone Interpreter. This patch is implemented by the following statement:

```
headers ] here [ dict org heads ##p! org ]
```

This is conditionally compiled depending on the setting of the “tethered” constant set earlier in the Job file.

For tethered operation, the dictionary is not required but some utilities that facilitate number entry are conditionally compiled, as follows:

```
:m #  number emit-s m;
:m ## [ dup 8 rshift $ff and swap $ff and ] # # m;
```

Note that it is possible to compile a Standalone Interpreter but debug definitions interactively with a Tether. In this special case, the dictionary location can be patched after the number utilities are defined.

Build Statistics

At the end of each Job file the following sequence executes:

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

This provides a post-compilation report of how much memory is used by the application, saves the programming image files (**chip.hex** and **chip.bin**) and display’s the Host PC’s stack.

One key purpose for displaying the Host stack is to reveal compilation errors. ***Any items left on the Host stack indicate a compilation error.***

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 a typical compilation report generated by executing the **c** command:

```
C:\work\jay\co\branches\private\projects\WARB4>C:\PROGRA~1\gfor
th\gforth.exe -e "fpath path=
./|c:\work\jay\co\trunk\source|C:\PROGRA~1\gforth" -e ".fpath" -e " 2
value com? " -e " 13 value current-baudrate "
c:\work\jay\co\trunk\source\amr\serial-windows.fs ./job.fs -e 'open-
comm target talking'
./ c:\work\jay\co\trunk\source C:\PROGRA~1\gforth

HERE=2546
Host stack= <0>
Talk to the target
```

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 the Job file, as described earlier.

After the "noise" of the command file echo, the most important parts are the amount of flash program memory used by the application and the Host stack contents.

For this example, the application ends at hex location $2546 (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, the MyForth system code, debug utilities (if loaded) and your application code.

The Host stack shows that there are zero items on it, as is required for a successful compilation.

After compiling (or downloading with **d**) you can verify communication with the Target by entering **.s** (print stack) to request the Target to display its stack contents (e.g., **<0>**).

You may also want to execute **words** at the MyForth command prompt to display the words that were just compiled. You can enter the commands listed by **words** at the MyForth command line and they execute on 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. This is because your project code is already stored in the Target's Flash memory, 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. Also, you can use **c** to recompile the application and disassemble some code: the Target does not have to be active.

If you are using the tethered interpreter, 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 "go" turnkey Word, your processor will automatically start up executing the turnkey Word.

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 with a Standalone Interpreter and a dumb terminal program. A later section explains this process in more detail.

Here are some compiling facts:

1. The **c.bat** file calls **gforth.exe**, which includes the Job file. System configuration and compilation are then controlled by the Job file.

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. This file can be used with the SL debug adaptor and IDE to program a chip. Note that GVim 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 also be examined directly with GVim.

3. The 2546 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>** at the MyForth command prompt, 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.

The disassembly of the definition is displayed one line at a time. To display the next line, enter **n**, the space bar, or any key except the terminator keys. The terminator keys are **q**, escape (**Esc** key) or the **Ctrl-c** key combination. The display will continue until you reach the end of the processor memory (whew!) or until you terminate it.

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  ;
```

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.

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. This provides in-line insertion of code sequences: 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, observe 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).

sees

It is sometimes more useful to see a specified number of lines without having to press keys to make additional lines appear, as is needed with **see**. Also, it is often necessary to generate a listing of a decompiled word. MyForth provides the Word **sees** to do both of these tasks.

When entered at the MyForth command prompt, the format of **sees** is:

n sees <word>

In the above, "n" is the number of lines to decompile and "<word>" is the Word you want to decompile.

A more common use of **sees** is to generate a listing. You can do this from a Windows command prompt by using the **sees.bat** command script. It compiles the current application and displays the decompiled Word in the Windows command window.

Normally, you will want to redirect the decompiled output to a file. Here is an example showing how to execute **sees.bat** from a Windows command prompt, displaying 25 lines, starting at the definition for "init" and redirecting it to a file named "init.txt":

sees 25 init > init.txt

You may first want to execute sees without redirecting the output to a file to make sure that the listing is what you want. Generally, you will need to adjust the number of lines to get the listing you want.

When executing from the Windows command prompt, the parameters for **sees** come after the command (in the normal way for batch scripts).

CAUTION: A Word to be decompiled with **sees** must not contain a file redirection operator such as ">" or "<" .

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 "$" (or "\$ for Linux users).

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**.

For example, assume that 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 starts 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.

Dump

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 (not Windows) command prompt.

For example, to dump starting at $0123, enter: **$0123 ## d**. The **d** command works in much the same way as **see**, outputting one line at a time, but only accepts an "n" to display the next line of the dump.

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

Scripts

Basic

You can execute commands that you would normally enter on the MyForth command line by putting them in a text file and executing them from a Command Prompt with the **run** command (also 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” and then 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.

Advanced

Advanced users may wonder why the sequence “0 do cr n loop” appears just after the right bracket. 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.

Because **do … loop** is not a MyForth looping construct, it may seem 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**).

Downloading

To compile and download your 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 Target's baud rate is set in the bootloader file (e.g., **bootloader300.fs**) and the GForth download rate is set in the **d** and **c** command files. The normal baud rate for MyForth applications 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).

Also check to see if the com port specified in the **c** and **d** files is the same as the PC's com port. With USB-based serial adapters, the com port can be just about anything. To verify the com port, use a terminal program such as Putty with a hardware wrap of pins 2 and 3 of the serial connector. If the com port specified in the terminal program echoes characters through the wrapped pins, then the com port number is the same as the terminal's com port.

As the **d** command executes and the download proceeds, you will be prompted for actions at each stage (e.g., press and hold 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 (try executing ".s" to see if there is a response).

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

If you are still having trouble with the download, verify that the Target contains a bootloader. If it doesn't, use the Silicon Laboratories debug adapter and IDE to program the chip with the **chip.hex** file from an application that uses the same Target processor. If your application compiles error free with the **c** or **d** commands, the **chip.hex** file in your project directory can be used to initially program the Target.

To work with MyForth, every Target chip must have a bootloader programmed into it.

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 command line followed by a **#** sign. The **#** 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 the Target executes the Word.

To enter a 16-bit value (e.g., an address), follow the number with **##** (also a Target 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. It also allows you to specify which stack is being referenced (i.e., Target or GForth).

Stack Display

As with most Forth systems, you can display the stack contents with **.s**. Entering **.s** at MyForth's ***ok*** prompt is a convenient way to verify that the Target is responding.

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 still be able to execute some commands from the command line – these are usually Words, such as **bye**, .that are defined in GForth.

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.

Serial port hang-ups seem to be more prevalent when using USB to serial adapters. Some are almost unusable. If possible, use a port such as COM1 or COM2 that may be tied to your PC's internal hardware and do not have to interface through a USB link.

Turnkeying

To turnkey a compiled and tested application, edit the **job.fs** file so that the application starts up executing the turnkey Word 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, 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**).

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:

- Memory and stack mapping
- Forth implementation
- MyForth programming

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

Memory

Overview

The following sections describe the mapping, allocation and access for the following types of memory:

1. **Processor RAM** – This is RAM that can be directly addressed by processor instructions without the use of a 16-bit data pointer. Because of this direct addressing capability, data bytes in this block of RAM are often called “direct cells.” The first 8 direct cells are special processor registers. Their use is explained more fully in the Register section.

2. **Flash** – Flash memory consists of non-volatile processor memory used to store programs and data. For processors with a large amount of Flash memory, such as the C8051F120, this memory may be addressed as “banks” of memory with bank switching controlled by special processor registers. This bank switching is not directly supported by MyForth (as it is processor specific), but code examples are available in the Project Directories.

3. **XRAM** (External RAM) – XRAM can reside either in the processor chip or in an external RAM. When XRAM is contained within the processor chip, it is generally accessed by dual mapping. When XRAM is truly external, it is accessed using port pins as address and data registers.

Generally, for each type of memory, memory access examples are provided. However, these examples use MyForth Words that are not covered until later sections.

The examples for each type of memory are primarily intended as a quick reference for users who are generally familiar with MyForth but need a refresher on how processor memory resources are mapped, allocated and accessed.

If you are reading this manual for the first time, you may want to skip these examples until you are more familiar with MyForth.

Processor RAM

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

Note that Processor RAM is also referred to as “direct memory” and bytes within this RAM are often called “direct cells.” This is because the RAM can be directly addressed. It should not be confused with flash memory or “external” memory (XRAM).

The direct cell 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 direct RAM, but MyForth supports processors with as little as 128 bytes of RAM.

The first 8 cells of direct memory are used for processor registers. Registers 0 and 1 are used for the “A” or address register and for the Forth stack pointer. Other registers from 2 to 7 are commonly used for looping and are generally not used for general purpose memory.

Direct cells from 8 to the top of the return stack are available for general use and are often used where access to XRAM is not desirable. The stack locations and use of XRAM are discussed below.

Refer to the Registers section below for more information on the use of the first 8 direct cells.

Boot Loader

The MyForth Boot Loader is located in low memory, just past the relocated interrupt vectors. A typical start address for bootloader code is location $008B. The bootloader requires approximately 240 bytes of memory.

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

MyForth programs are stored in Flash memory starting just after any re-mapped interrupt vectors. This will be just past location $200 (512) in most Silicon Laboratories chips or just past location $400 (1024) on the C8051F120. A typical start address for program code is $022D.

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 programs will start just after MyForth system definitions.

Stacks

The location of the data and return stacks varies, depending on the number of direct cell RAM that is available.

For chips with just 128 bytes of RAM, the return stack starts at $21 (33), 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 (127), also to leave room for variables and bit variables, and grows upward (increasing addresses) toward the data stack. For these chips, the data stack starts $fe (254) and grows downward toward the return stack.

Variables

As mentioned above, the area from the top of the return stack to register 7 (e.g., 7 to $7e or approximately 120 bytes) is available for general use such as variable allocation. Variables are allocated using the MyForth Words **cpuHere** and **cpuAllot**. Here is an example of how a double variable is allocated and used:

```
cpuHERE constant my2var 2 cpuALLOT  \ allocate two direct cells

: @my2var  ( - lsb msb)    [ my2var 1+ ] #@   my2var #@ ;
: !my2var    ( - lsb msb )   my2var #!  [ my2var 1+ ] #! ;

: ?my2var   ( - adr )  my2var #  ;    \ see where it was allocated
```

In the above example, note how the left brackets are used to invoke GForth to calculate the address of the most significant byte (a typical is value is 8).

XRAM

Many of the Silicon Laboratories (SL) processors support “external” or XRAM. On many of the processors, this “external” RAM is located on-chip and mapped on top of processor memory starting at location 0.

Often, a large amount of memory is needed for structures such as arrays or buffers and it is not desirable to use the limited amount of direct memory available between location 8 and the top of the return stack. In these cases, XRAM can often be used to preserve variable space.

A relatively large amount of on-chip XRAM is generally available on SL processors. For example, 1024 bytes of XRAM is available on the C8051F310 and C8051F362, 2048 bytes on the C8051F410 and 8K (bank addressable) bytes of XRAM are available on the C8051F120.

This on-chip XRAM is addressed using the **movx** instruction after loading the data pointer, **p**, with the desired address.

Some chips may also support addressing of true external memory using two of the 8-bit I/O ports for addressing. This is also accessed via the **movx** instruction but the data pointer requires special configuration (see the applicable Silicon Laboratories manuals).

There are no MyForth words that directly support external addressing but it is relatively straightforward to implement definitions that can handle external addressing.

The data pointer is set with **!p** and **##p!**. Once the pointer is set, data can be written or read with **!x** and **@x** .

To read or write sequential addresses in XRAM, use the **@x+** and **!x+** instructions: they automatically increment the data pointer after a byte has been fetched or written.

Here are some examples showing how to access XRAM data starting at location $22:

```
: @$22  ( - n)    $22 ## p!  @x ;
: !$22    ( n -)  $22 ##p!   !x ;

$22 constant start-byte

:m mybytes    start-byte ##p! m;

: @mybytes    ( - n1 n1 n3 n4)   mybytes  @x+  @x+  @x+  @x+  ;
```

The disadvantage to using XRAM over using variables allocated in direct cells is the extra code and time needed to set the XRAM address.

Flash

Flash memory is normally used to store your program or to store static data. However, Flash can be written, using the Words **+write** and **–write** to enable and disable Flash access. Note that flash bytes must be written into blocks of memory that have first been erased. Otherwise, access to bytes in flash is similar to that used for XRAM.

Here is a code snippet from the C8051F410 bootloader showing how the program image is written to flash:

```
\ --- accept bytes and write them into flash, starting at $200
        .
        .
        $200 ##p!
        begin
                0 # 6 #for
                        'KEY +write (!x) -write p+
                        'KEY +write (!x) -write p+
                6 #next
                7 (#@) 'EMIT
        7 #next
```

Note that the flash block must first be erased before writing data to it. Here is an example of how to save and restore from flash (for the C8051F362):

```
cpuHERE constant interval-choice 1 cpuALLOT   \ allocate direct cell

\ note: on the 362 chip, the 1K block starting at $7C00 is reserved!
$6C00 constant config-block  ( config page is $6C00 to $7000)

\ note: in erase mode, writing any byte to the block erases it.
\ Write $ff so bits are 1 (once zero, it can't be set back without erase)
: erase-config    config-block  ##p!   $ff # +erase  !x  –write ;
: save-interval
     interval-choice  #@   config-block ##p!  +write  !x -write ;
: get-interval    ( - n)      config-block  ##p!   |@p ;
: restore-interval    get-interval   interval-choice #! ;
```

Flash can also be used to store static data. The data can be allocated when the program is compiled into the program image. Allocation is performed using **here**, **org** and **allot**. Data is compiled in the Flash image with "," (comma).

Here is a simple example of creating and accessing a two-item table in Flash:

```
: table-start ;  \ use with "see" to look at table entries

here   constant   flash-table 2 allot   \ allocate space for table
       (definitions)
       (more definitions)
here   \ put current compilation pointer on the stack
flash-table org    \ set compilation pointer to start of flash-table
$aa ,  $bb ,       \ store the data in the program image
org    \ restore current compilation pointer
```

This example may seem a bit cryptic. Here is what is happening:

1. The table-start definition is a dummy to be used with "see" to look at the data that has been compiled in the table (it is optional for debug).
2. The second line assigns the current compilation pointer to the constant "flash-table" and moves the pointer two locations forward using "allot"
3. After a number of program definitions, "here" puts the current compilation address on the stack so that it can be restored.
4. The next instruction sequence sets the compilation pointer back to the table address (i.e., with "flash-table org") and compiles ("commas") two bytes of data there.
5. The last "org" instruction takes the compilation pointer savedon the stack and uses it to reset the compilation pointer.

Here is a definition to read the contents of the table:

```
: @flash-table  ( - n1 n2)   flash-table  ##p!   |@p+  |@p  ;
```

Implementation

Threading

MyForth is a subroutine (call) threaded Forth. Note that many named sequences are defined as macros (with the defining Word “m:”) and are 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 performs the return. This optimization saves memory and increases speed. This is efficient tail recursion, but also works as a “goto” (jump with no explicit return).

Vocabularies

Don’t skip over this section!

Very much like early ColorForth, 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 (GForth) 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.

Recommended MyForth programming practice is to use these two Words explicitly when the intent is to find Words in the Host (GForth) or Target vocabularies.

However, this is just a recommended practice. If a Word is defined only in GForth or only in the Target vocabulary, there is no harm in searching both.

If you examine the source for MyForth, you will see that **[** and **]** are used in a variety of ways to flexibly reference either the GForth compiler or the MyForth Target compiler.

These two Words 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 **[** and **]**, 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 these 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 on the PC exercises these Words interactively, but MyForth also allows you to put the name headers on the Target to build a Target that has its own standalone interpreter. This type of operation is described later.

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 has a name. When this name appears in a definition, the macro executes immediately to compile instructions or data into the Target image.

The Target image is a block of memory residing on the Host starting at the location named **target-image**. It is an image of the bytes that are compiled by the **c** and **d** commands. This image is 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 sequence 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** is loaded after your application definitions are 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 you may 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. The MyForth designation for 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, put **#** after the number. The action of **#** Word is to use a number on the GForth data stack to create code that will put the number on the Target's data stack when the definition is executed (i.e., **#** compiles a literal).

To put a bit inverted version of your constant on the stack, use **~#**. This Word is often used to perform logical operations on Special Function Registers, setting and clearing individual bits.

The Word **##** will put a 16 bit literal, often 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**. In the **set5** example, the use of **(#!)** instead of **#!** would leave $23 on the Target's stack.

Stack Initialization

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. The return stack is 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 MyForth 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. *Note that 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 that is used to indirectly access data.

You can use **a!** to load **a** with a value (e.g., register number), 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 **!+** can be used 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!  @+ @+ @  ;
```

Note that the macros **(!)** and **(!+)** are variants of **!** and **!+**. Like most MyForth macros defined within parentheses, they do not drop the stack after they complete.

This behavior is useful during operations such as initialization, where the contents of the stack does not need to be preserved.

For example, this is how to initialize a four-byte data array, starting at direct cell location 8, to a value of $A5:

```
8 constant array   \ start array at direct cell 8
: init-array   array # a!   $A5 #   4 #   7 #for  (!+)  7 #next ;
```

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 directly sets **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**.

Here is a simple example showing how to use **##p!**:

```
here constant choice $30 ,  \ embed value for choice in flash

: get-choice  ( - n)   choice ##p!   |@p ;
```

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 definitions are not very useful and you would not typically use them 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 using 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:

```
\ -----[ variables and constants ]

$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**, 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 (GForth) and acts as a label for the start of **mycells**. The "8 cpuALLOT" allots 8 cells after **mycells** by moving the **cpuHERE** pointer forward 8 bytes. Of course, you must use **mycells** within a Target definition if you want to use it on the Target. For example:

```
: ?mycells  ( - n)   mycells # ;
```

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 # ;   \ put a 7 on the stack, representing a direct cell adr
```

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 (GForth): 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.

Normally you would use **##** to put the double number on the Target’s stack in the correct order, but this example illustrates how you can manipulate data on the Host before using it for a Target definition.

Definitions 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 Job file, when you define a new interrupt.

The application's interrupt vectors are defined near the top of the Job file using statements such as "$208 constant TIMO" which establishes the remapped vector address for the Timer 0 interrupt.

Editing the Job file to add a new interrupt vector and changing the location of **rom-start** 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 start at $200; for the C8051F12x and C8051F362 family of chips, they start at $400.

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

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

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

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

The **start** before **interrupt** 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. 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.

Note that the definition for cold that follows interrupt is the actual code that will be executed when the interrupt arrives.

The example above shows how the "cold start" interrupt vector is defined and installed. For this example, **start** is defined as the location of the remapped interrupt vectors at location $200 (see any Job file for a 300 processor).

If you look at the compiled application you will find a jump to the code for **cold** at location $200 (start). You can examine the code at **cold** and note its address by entering "see cold" at the MyForth prompt.

Conditionals

Overview

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

In MyForth, the **if … then** construct uses the value in **t**, the top of stack, as you would expect, to conditionally execute code. MyForth also provides special conditionals such as **if', 0=if', if.,** and **0=if.** to conditionally execute code based on bits or carry being set or clear in **t**.

The following sections describe each of these and the code they produce in more detail.

Note that MyForth does not provide an "else" conditional. By examining some MyForth system code and application examples, you will see that it is not necessary.

Also, there are not 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 MyForth concepts. 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 you can see why "else" is not part of MyForth: there are ways to code so that it isn't needed.

Next, look at the **drop** in front of $0a and $0b. Like the examples with **until** (next chapter), **t** is not automatically dropped. Another way to say 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 rather than just exiting the definition.

One other reason for not consuming an 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**, and you may need to save it (e.g., the carry bit) if it is needed in later processing. It is much simpler and clearer to jump on the state of **t** and clean up later: if the value that caused the jump is needed, there is no need to get it. If it isn't needed it can be dropped.

What about **0=if**? Most of the "if" 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 executes code based on a bit being set. Here is a reworked version of **iffy1** based on checking a bit.

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

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

```
---------- iffy2
081B  30 E1 05      jnb ACC.1,0823  if.
081E  18            decR0  (dup
081F  F6            mov @R0,A  dup)
0820  74 0A         mov A,#0A  #
0822  22            ret  ;
0823  18            dec R0  (dup
0824  F6            mov @R0,A  dup)
0825  74 0B         mov A,#0B  #
0827  22            ret  ;
```

The previous definition is a contrived example. Usually you would be checking and branching 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 executes code if the carry bit is set. Here is a simple example that always puts $0a on the stack:

```
: iffy4  [ clrc ] $80 # 2*' drop if' $0a # ; then $0b # ;
```

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  clrc $80 # 2*' drop 0=if' $0a # ; then $0b # ;
```

The code compiled by **iffy5** is similar to that shown above for **iffy4**, 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 a 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 **iffy6**, you will see that putting a positive number such as $7F on the stack and executing **iffy6** will result in a $0b being put on the stack. Try entering the following at the MyForth prompt:

```
$f7 # iffy6 .s
```

This will put $0a on the stack.

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

=if and **<if**

The **=if** and **<if** conditionals are used differently from the conditionals described above.

These two conditionals require a literal value, supplied at compile time, for the comparison value; they do not operate on a stack value supplied at run time.

Here is an example of a definition that correctly uses **<if**:

```
:m  |1char  ( n - n')  dup 48 # <if drop !err ; then drop adjust  m;
```

Note that a “bar” symbol precedes “1char” in the above definition. There is nothing mystical about it: it is just a convention used in many MyForth definitions to indicate that the definition is a macro, not a Word. Because macros cannot be executed interactively, you will often see a definition like this:

```
:  1char  ( n - n')  |1char  ;
```

Typically, you would use a definition like the one above to interactively test the macro from the console with values supplied from the stack (e.g., via **#** or **##**).

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>** is the register or direct cell to 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 execute 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,#32  #!
0706  DD FE         djnz R5,0706  #next
0708  DE FA         djnz R6,0704  #next
070A  DF F6         djnz R7,0702  #next
070C  22            ret  ;
```

Conditional

In MyForth, conditional loops are formed with a **begin … again** 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.**.

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

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

again

Using **again** as the loop conditional forms 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.

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

The definition of again is:

```
:m again  call ; m;
```

Note that the **;** after the **call** is used to optimize the **call** to a **jump**, if possible. Thus, **again** can be used alone, to perform this operation, as illustrated in **bootloader120.fs**.

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 **begin … 0=until** . 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

Like the =if and <if conditionals, **=until** and **<until** (see below) require the comparison value to be specified when the definition is compiled.

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 used with the **=until** and **<until** conditionals.

<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, 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 the accumulator is negative (i.e., the most significant bit 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
```

The above code 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 "!" that 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.

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**.

These Words are useful in manipulating the least significant byte of a double number (for example) but do not do a full 16-bit increment or decrement.

negate and **invert**

MyForth does not have a "-" Word to subtract two numbers but two numbers can be subtracted by negating one number an adding the two together.

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 two unsigned bytes on the stack, leaving the double precision (16 bit) result on the stack with the 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** performs a divide operation on 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. Note that **three** and **two** in the definition of **(u.)** are followed by semicolons which normally compiles a **ret** (return instruction). 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. It provides most of what you need and, if you need 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." Please 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 tools are flexible enough, once they are 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 Words consist of a named set of assembly language statements and/or macros that define a Forth Word. In such an implementation, Code Words would start with a Word such as “code” and terminate with a Word 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 MyForth provides some useful assembler definitions, 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 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 "in line" approach to efficiency is seldom needed. As mentioned earlier, MyForth is primarily a macro assembler implemented within a Forth conceptual framework. Thus, you are always in an "assembler" environment.

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

This is also what macro definitions do. In fact, it is often unnecessary to use brackets to denote assembler definitions; this is recommended practice to clarify the programmer's intent but you may have observed that it is omitted in many of the MyForth system definitions.

One common example is the "nop" instruction – there is no MyForth Word with that name so, when it is invoked, it is "found" as an assembler definition. Thus, it may be used outside the brackets usually associated with an assembly code sequence.

But, some caution must be observed: there are several assembler definitions that have the same names as MyForth Words. These include **push**, **pop** and **swap**. These are 8051 assembly language instructions and are also MyForth Words that manipulate the stack.

The use of the assembly language versions are explained in the following sections.

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.

In the Compiler chapter, the section describing the return stack 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. In the MyForth definitions, the assembler versions of **push** and **pop** are applied specifically to **t**, the top of stack, and thus push and pop only the top of stack.

In the assembler, **push** and **pop** can be used with *any* direct cell; in MyForth, **push** and **pop** apply *only* to the top of stack.

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 and executed in the Host vocabulary.

The format shown in **set-example1** is perhaps preferable because it shows the intent of the coding. It is also a safer form to use when you are not sure whether or not a Word is defined in both vocabularies.

The "1" in the above just puts a 1 on the Host's stack which is used by the following definitions to assemble instructions to set bit 1 of the top stack item. Outside the brackets, 5 also goes on the Host stack, but the Word **#** following it assembles code that puts a literal 5 in **t** *when the definition is executed*.

Pins and **Bits**

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

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.

Here is how to use **set** and **clr** to change port bits:

```
: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 lay down code in the Target image.*** In the **enable** example, “3” puts a number on the Host’s stack and “.P2 clr”, executed by the Host, assembles instructions in the Target image.

Here is a more detailed description of how the bracketed instruction sequence (shown above) operates:

1. The left bracket establishes Forth as the first vocabulary in the search order,
2. The “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 on the Host PC,
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 the first vocabulary in the search order.

To see what is compiled, use the **see** command to decompile 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; in fact, *all numbers not followed by a # will be put on the Host's stack*. The **.P2** converts the number on the Host's stack to a bit address appropriate for use by **clr** and then puts 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.

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 and can't be decompiled by see – they simply compile code in the Target's image (on the PC) 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 examples show how to use **mov** 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 in carry to a bit address.

Here is an example:

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

Note that the definition is a macro defined using the assembler but it is not bracketed. This is because **movbc** and **movcb** are not MyForth (Target) 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.

Putting the “3 .P2 movbc” sequence within brackets would 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;
```

Again 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 #@  ;
```

reti

The **reti** macro assembles a return from interrupt (reti) instruction.

6

Boot Loader

Overview

Understanding the Boot Loader (bootloader) is not needed for normal use of MyForth. The information in this chapter is provided to help you understand more about the operation of MyForth.

Purpose

The Boot Loader's sole purpose is to do basic initialization of the chip and download your application code into the processor's flash memory. Thereafter, it is no longer used.

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** (d.bat) command. This command compiles your application and downloads it to the chip.

Note that the only difference between the **c** command and the **d** command is that the **d** command downloads to the chip after compiling the application. Thus, if you do not need to download to the chip, you can immediately interact with the chip after recompiling with the **c** command.

This is typically what you will do after powering up your chip to start a new programming session. In that case, your program is already stored in the chip's flash and all you need to do is re-establish the connection to the chip.

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 or USB Debug 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

Overview

All MyForth boot loaders are coded to be compatible with the AM Research Boot Loader. Although AM Research does not provide standard support for the C8051F12x or C8051F41x chips, the MyForth Boot Loader uses the AM Research handshaking protocol for its native Boot Loaders.

In the future, as MyForth adds native boot loaders, they will continue to be compatible with the AM Research Development System

For chips supported by a native MyForth Boot Loader, you can use the **chip.hex** and **chip.bin** files in your application directory to establish the Boot Loader. These files are written each time you compile your application with the **d** or **c** commands.

Installation of the MyForth Boot Loader for Silicon Laboratories chips requires a hardware debug adapter and the IDE software furnished with a development system from Silicon Laboratories, as described below.

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 with an AMR Boot Loader already installed: you can use one of these with MyForth without any additional programming.

The AMR Development System also has a JTAG/C2 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/C2 interface is adequately covered in the AM Research Reference Manual and is not covered here.

Silicon Laboratories

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

1. An EC-2 Serial Adapter or a USB Debug Adapter connected to a 10-pin JTAG/C2 interface connector on a Silicon Laboratories Target Board (or wired to the JTAG pins of your own Target processor)
2. The Intel HEX download function of the Silicon Laboratories Integrated Development Environment (IDE).

The EC-2 Serial Adapter or USB Debug Adapter and IDE are furnished with Silicon Laboratories development systems. The use of the Intel HEX download function from the IDE is straightforward: just bring up the IDE and select the appropriate menu options.

The following page provides a detailed procedure for downloading.

If you have purchased a development system for a newer chip, such as the C8051F410, then Silicon Laboratories will provide a newer IDE and a USB Debug Adapter. We recommend using the USB Debug Adapter, if you have it.

Here is a detailed procedure for downloading the Boot Loader using either the EC-2 Serial Adapter or the USB Debug Adapter and the Silicon Laboratories IDE:

1. Connect a serial port to the EC-2 or a USB port to the USB adapter and connect the JTAG/C2 cable to the 10-pin JTAG connector on the Target Board (it is the 10-pin shrouded connector).
2. Power up the Target Board (power is not required for the EC-2, although a power connector is provided).
3. Bring up the IDE. On an older IDE such as V1.85, select the Options/Debug Interface menu items and then select either the JTAG or C2 (Cygnal 2-wire) option, as appropriate. For a newer IDE that supports the USB Debug Adapter, select the Options item on the Options/Connections menu and then select either the EC-2 or USB Debug Adapter. On the same options panel, also select either the JTAG or C2 interface, as appropriate (e.g., JTAG for 12x chips and C2 for 41x chips).
4. Select the Debug/Connect menu option (this is the same for older and newer IDEs). The connection to the processor should be indicated on the bottom of the screen.
5. Select the "Download code" option by clicking on the "DL" icon on the toolbar or by pressing Alt-D (this is also the same for both IDEs).
6. On the dialog box that will pop up. For an older IDE, select the "not banked" option. Then, select the "Erase all code space" option (probably not required, but …)
7. Select the Browse button to browse to the directory containing the **chip.hex** file you want to download. This can be any application directory for that processor, provided that the **c** or **d** commands have been used to compile the application
8. You may also have to select the file type you are looking for (it is Intel-HEX

Once a program is downloaded to the chip via the JTAG/C2 interface, the EC-2 or USB Adapters and the IDE are no longer needed.

Note that when a chip.hex file is programmed into a Target processor, the target processor will have a bootloader – all chip.hex (and chip.bin) files contain a bootloader in addition to application code.

Operation

Location

For Silicon Laboratories chips, the Boot Loader resides in the first page of flash RAM starting at $0000.

Overview

The Boot Loader consists of a very small amount of code (about 240 bytes) that performs the following:

1. Sets up basic chip resources such as the watchdog interrupt, cross bar, oscillator and serial port
2. It establishes new interrupt vectors to supplant those on page 0.
3. Checks to see if there is an active download request from the Host and, if so, downloads code from the Host over the serial port
4. After downloading or after a timeout period, it jumps to your application (turnkeyed systems) or to the Target's tethering software (interactive development).

Interrupt Vectors

The Boot Loader re-maps the interrupt vectors to Page 1. For most Silicon Laboratories chips, these the re-mapping starts at location $200. For the C8051F12x chips the vectors are mapped to $400 because of the 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 remapped interrupt vector (see **rom-start** in the **job.fs** file for this location). In some cases, if there are no other interrupts used, this will be immediately after the Cold Start vector.

Startup

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 starting at location $200 or $400 (e.g., **decode $0200**).

The bootloader source code is contained in files such as **bootloader300.fs, bootloader310.fs, bootloader410.fs** and **bootloader120.fs.** This code is "included" in **job.fs** (e.g., see job300.fs in the \MyForth\chip directory). Thus, you can examine the remapped vectors starting at $0000, the bootloader code (starting around $ab) and the application, starting just after the remapped vectors at $200. We suggest you compile an application and examine code at these locations using the **see**, **sees**, or **decode** commands.

You can examine the bootloader code by entering **decode $0000** (you might also try entering "see boot"). For any system, however, you can examine the actual content of these locations using a dump.

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

Example

You can see bootloader code by examining the bootloaders in the MyForth\chip directory (e.g., the bootloader120.fs or bootloader300.fs files).

Advantages

To some, the overhead of the bootloader and the debug definitions that MyForth may optionally load seems wasteful of processor resources. We feel that the small overhead is worthwhile. The following sections provide more perspective on the use of an interactive environment.

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 or JTAG debugger.

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.

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.

Because of the multiple use of these routines, it is unlikely that they have any hidden bugs: they are used by various routines before you select them for re-use: if they were not reliable, your application would not work reliably.

Reduced Program Size

You will observe that programs do not grow very fast after a certain point. This is 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 make your programming easier.

For non-trivial applications re-use of code can significantly reduce the size, reliability and coherency of your application; for small programs, perhaps using macros for speed, program size is not an issue.

7

Tethered Target

Overview

This chapter describes how the MyForth tether to the Target processor works. It is provided for those who wish to know more about this important function.

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

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 then 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.

Target Interpreter

The following describes how the Host interacts with the Target processor to implement an interactive Forth test environment. The source code for the Target Interpreter (tethered interpreter) is contained in **tether.fs**. Because the Target only needs to be able to execute code at a specified address, the required definitions are few and deceptively simple. Here is the entire source code listing for the Target Interpreter:

```
] \ 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 ;
```

Following sections describe the operation of the two fundamental definitions in this listing, **execute** and **quit**.

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**).

Development with the Debug Adapter

In some cases, you will experience problems downloading via the serial port and the bootloader. Most of these problems are caused by USB to serial adapters. This is especially true of adapters that buffer data and thus interfere with bootloader handshaking.

If you have trouble downloading over the serial port, you can develop interactively using the Silicon Laboratories USB Debug adapter and the IDE. This still requires the use of the serial port but the timing requirements are more relaxed.

For example, to develop using the IDE and a Silicon Laboratories Target Board, connect the USB Debug Adapter to the board and use the IDE to download the chip image (in **chip.hex**) to the chip. Note that you must first compile a current program image in **chip.hex** using the "c" command.

To do this, select the "Connect" option on the Debug menu. This should connect the Debug Adapter to the chip. When connected, the target connection pane at the bottom of the window should (for example) change from "Target: ????????" to "Target: C8051F300" to indicate that the adapter is connected. The code image of the target at the right of the window should also indicate memory contents.

Once the chip is connected, use the "Download Object Code" option on the Debug menu to download the **chip.hex** image to the Target. After downloading, disconnect from the Target using the "Disconnect" option on the Debug menu.

To begin interactive development via the serial port, reset the Target and bring up the tether with the "c" command. You can verify that the Target is talking via the tether by executing ".s" – this should produce a display of the Target's stack (e.g., 0>). To further confirm interactive operation, try putting a value on the Target's stack (e.g., "5 #") and displaying the stack – the value should be displayed on the stack after executing ".s" a second time. Executing "." removes the value from the stack and displays it.

Interactive development via the serial port can continue normally once the Target is talking. Note that the method outlined above will work only if the serial port is operating and is intended as a workaround for serial adapters that cannot reliably interact with the downloader.

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

A dumb terminal is needed to exercise the standalone interpreter. To make it easier to test the Standalone Interpreter, MyForth provides a dumb terminal that can be executed from a MyForth command line. This terminal program, **dumb.fs**, is loaded in **loader.fs**. Of course, you can use another terminal program, such as Putty, to exercise a Standalone Target; in this case, you can comment out the line that includes the MyForth dumb terminal.

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 cannot 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 consist of 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 describes 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 shown below. The descriptions in this chapter are based on this listing.

Basic Definitions

First, observe that some Words are defined with"-:" instead of ":". These "dash" Words are callable by other Words, but their headers are not compiled on the Target. Although these Words cannot be executed interactively from a terminal their advantage is that they do not up any dictionary space on the Target.

The listing starts off by defining some basic building blocks for character I/O such as **key**, **emit**, **echo**, **space**, **cr** and **ok**. These Words perform the same functions as in any standard Forth system.

Similarly **2dup**, **clip**, **min** and **max** manipulate and clip stack items. These will not be discussed in detail, 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 that the stack pointer is **s**.

```
] \ 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 ;
```

tib

The start of **tib**, the terminal input buffer, is defined with a constant. It starts at register 2, just after the stack and address pointers, **s** (register 0) and **a** (register 1).

The first cell in **tib** holds the count for the Word in **tib** to be interpreted. As described in the section below on the dictionary, each dictionary entry consists of a count and the first three characters of the Word to be interpreted.

When **tib** contains a complete entry the first four values have the identical structure. Thus, for a complete entry, the first **tib** cell (register 2) contains the Word count and the following three cells contain up to the first three characters of the Word to be interpreted.

Note that there are five cells in **tib**; the last cell is used to store the last character entered in excess of three characters. This avoids having to handle excess characters: they are always stored in the last cell of **tib**.

To fill **tib**, **query** calls **0tib** to initialize **tib**. Here is how **0tib** works:

1. The location of **tib** is put on the stack (**tib #**), duped and stored into the address pointer (**a!**). This sets things up for storing five zeroes into **tib**.

2. Five zeroes are stored in sequential **tib** locations with “0 # dup !+”. Note that “!+” stores a value in the cell pointed to by the address register and increments the address register. The sequence ends with a simple **!** (store) because there is no need to increment after the last store.

3. Lastly, the address of **tib** is restored in **a**.

The other **tib** Word to be explained is **tib!**. This takes a character from the stack and stores it in **tib**, updating the **tib** count. Here is how it works:

1. After preserving **a** with "a push", the count is updated with "tib #@ 1+ tib #!".
2. Next, "tib # dup a! @" is used to get the updated count, saving the address of **tib** on the stack with a "dup". The "+" adds the address of **tib** to the count to obtain the address of the next available location, using "6 # min" to clip the address so limit the number of characters stored in **tib**.
3. The calculated address is stored in **a** with **a!** and a subsequent **!** stores the current character. Note that the clipping ensures that any characters entered after the first three characters is sloughed by storing it in the fifth tib cell, which is not used in dictionary matching (i.e., by **match**). The "pop a!" restores the previous contents of **a**.

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, another **xor** recovers the character, storing it in **tib** and returning 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 the 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.

Dictionary

Location

The dictionary for the Standalone Interpreter can be located by first executing "see dict" and noting the address loaded into DPTR at that location. The decoding will look something like this:

```
---------- dictionary
02FF 90 03 CC      mov DPTR, #03CC   ##p!
0302 22            ret ;
```

Note that the definition of "dictionary" in **standalone.fs** compiles a "dummy" load of DPTR. This is patched later, in **job.fs** with the following code:

```
headers ] here [ dict org heads ##p! org ]
```

Structure

Each dictionary entry consists of the following:

1. The length of the Word's name
2. The first three ASCII characters of the Word's name
3. The MSB of the location of the Word's compiled code
4. The LSB of the location of the Word's compiled code

Using the example given above, entering "$02FF decode" will produce a sequence of bytes like this for the dictionary entry for "cold":

04 ← length of the Word
63 6F 6C ← first three ASCII characters "col"
03 ← MSB of Word definition
C4 ← LSB of Word definition

Entering "$03C4 decode" will confirm the vector does point to the "cold" definition.

9

Examples

Overview

The following example applications are all loaded by the **job.fs** file in their project directories. The Job file manages the loading of source code files that support the application such as the Special Function Register definitions and the bootloader.

The examples that follow concentrate on the code that implements the application. Because the support functions and application management are given elsewhere in this manual, they are not included in the following application descriptions.

The applications represent two processor extremes. The Pseudo Random Sequence Generator application, in the PSR Project directory, uses a C8051F120 chip to illustrate what a fast processor can do.

The LCD driver application, in the LCD Project Directory, uses a tiny and inexpensive C8051F300 processor to illustrate how it can function as a peripheral controller.

Random Sequence Generator

The code that follows is for a 32-bit Pseudo Random Sequence generator. It was coded for the Silicon Laboratories C8051F120 processor running at 98 MHz.

The code for this example is contained in the PSR Project Directory and the application is loaded by the **job.fs** file. The source code for the application is contained in the source code file **psr.fs**. We suggest that you take a quick look at both of these files before reading further.

Pin Assignments

Only two pins are assigned for the register. The random output is assigned to P1.6, which normally drives an LED. For fast random noise output, of course, the LED does not visibly change; in this case, the **outbit** pin can be viewed on an oscilloscope (or monitored on a crystal earphone connected to ground).

The Microsoft Word document PSR.doc, in the PSR Project Directory, provides a more detailed description of the application and some photographs of the PSR output captured on an oscilloscope.

```
\ 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; \ all clues disappear!
```

Note that **~P1.6** (toggle Port 1, pin 6) and **~P.7** are defined at the end of the application to help test it.

Shift Register Setup & Initialization

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

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

Shifting

The following definition for **psr** is the heart of the application, shifting the register once (from right to left in the diagram above) and applying the feedback bits to an exclusive or to generate the bit to be shifted into the lower end of the register.

Note that the feedback bits, 18 and 31 were selected because they produce a maximum length repetition cycles (2^{32}-1). How these are selected is beyond the scope of this document.

The code below is reasonably straightforward and the purpose of each of the operations are described in the comments. It may be instructive, however, to use **see** to examine the assembled code.

One thing that may be a bit confusing are the “sequence” phrases within brackets. The brackets are used to use the Host (GForth) to perform an address calculation for one of the four bytes used to form the 32-bit register. The **(#@) 2*’** gets the byte from the specified address and shifts it left one bit, accounting for the carry (this is why **2*’** is used instead of **2***).

Note that the clue bit is turned on at the start of the routine and turned off at the end of the routine. This provides a signal that can be observed on an oscilloscope to monitor the timing of the **psr** routine. See the **PSR.doc** manual in the PSR project directory for oscilloscope screen captures showing this signal.

```
\ 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;
```

Display

The **.psr** Word provides a way of looking at the contents of the register. Output is displayed in hex, unsigned decimal and as a double number, courtesy of **h.**, **u.** and **d.**, respectively. Again, note the use of brackets to invoke the Host to calculate byte addresses.

```
\ 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.
        ;
```

Initialization & Test

The **psr!** Word takes four bytes, represented by n1, n2, n3 and n4 in the stack diagram, and stores them in the register. Note how efficiently this can be done by putting the start address of the byte sequence, **sequence**, in the address register and then indirectly storing the bytes using **!+**, the auto incrementing store operator.

The **0psr** Word zeroes the register and then uses **?seed** to initialize it to a pattern of alternating ones and zeros. The **init** Word initializes the register and shifts it 256 times to start with a scrambled bit pattern. To test, use **t** to shift the register 13 times and then display the results.

To run continuously, use **go,** which initializes the chip's crossbar switch, initializes the register and then continuously executes **psr**.

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

\ Note that #@ and #! push and pop the data stack, but
\ (#@) and (#!) assume the top of stack is already free to be used, so
\ there is no need to push or pop.

: 0psr          0 # dup dup dup psr! ?seed ;
: init          0psr 0 # 7 #for  psr  7 #next ;
: t             13 # 7 #for psr 7 #next .psr ;

: go    init-xbr 0psr begin psr again

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

LCD

The following code shows how to use a C8051F300 chip to drive an LCD display.

The code for this example is contained in the LCD Project Directory and the application is loaded by the **job.fs** file. The source code for the application is contained in the source code file **lcd.fs**. We suggest that you take a quick look at both of these files before reading further.

Pin Assignments

The LCD Application starts with a block of comments describing how the LCD display is wired to the chip. There is nothing much to say about this except to note the use of the "0 [if] … [then]" conditional, defined in GForth, to create a comment block.

The comment block text is given below.

```
\ lcd.fs

\ Nibble mode LCD driver for C8051F300 -- 22Aug06 cws/rjn
\ based on working driver written in AMR Forth
\ SFR definitions are included in job.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]
```

Pin Configuration

The code below sets up the pin I/O for the LCD. The first thing to note is the use of] to set the Target vocabulary. This is 'belt and suspenders" programming but does illustrate how bracketing can be used to note the intended vocabulary context.

```
] \ Target Forth
```

Next is the I/O setup for the 300's pins (11 of them, total, including power and ground!). In this case, all pins will be used as outputs so the macro **push-pull** sets all bits in P0MDOUT to ones with **ior!**, the "inclusive or" function of MyForth.

Following definitions provide convenient macro names for ports, special function registers and output pins. The **instruction** and **data** macros set and clear bit 7 in Port 0 to signal either an instruction or data command to the LCD.

```
\ ---------- 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 ] ;
```

Delays

```
\ ---------- 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 ;
```

Character Output

The Word **lcd** is the heavy lifter in the LCD application: it takes a character on the top of stack, **t**, and outputs it to the display a nibble at a time. Note the difference in usage of **and** and **and!**: **and** performs a logical operation on two stack items while **and!** performs a logical and on a port register.

The **clear-lcd** Word shows how output to the LCD can be combined with signaling bits to perform a function, not display a character.

```
\ ---------- 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 low nibble
   $0f # and pins ior!
   strobe ;

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

Initialization

The **init-lcd** Word initializes the display while **init** ensures that the pins are correctly set up for output.

```
: 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 ;
```

String Output

The lcd-type Word outputs a string located at the double (16-bit) address on the stack. Note that the **@p+** gets the string's count, which is then used by the **begin … 0=until** loop.

The definition of **"** looks a little strange but once dissected isn't too bad. The "34 parse" just parses the input stream using a quote as a delimiter. The "here there place" phrase moves the parsed string from GForth's "here" to the Target image. The "here there [c@ 1 +] allot" phrase makes sure that enough room is allotted in the Target image to accommodate the string (simple, huh?).

The **string** Word gets the two bytes specifying the address of greet, arranges them in the correct order, and passes the address to **lcd-type**.

The **greet** Word, executes **string** to output the string compiled by the quoted phrase that follows.

```
\ ---------- 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
```

10

Troubleshooting

Overview

Because MyForth does not have much extra code for error recovery, it must depend heavily on the operating system (e.g., batch files), GForth and (you guessed it) the user. Because the source of some errors may not be obvious, the following discusses common problems and their resolution.

This section primarily consists of problems encountered and documented during project development. Because of this, it is not organized in any particular order (e.g., according to frequency of occurrence).

Terminal Errors

When compiling with the **c** and **d** batch files, an error message may appear that includes the following:

***the Terminal*:0: File I/O exception**

This normally indicates that an error has occurred before attempting to load the **job.fs** file. Examine the **c** or **d** batch files for files that they load. For example, a file such as **loader.fs** may be missing in a local directory or (more commonly) a path to a file contained in the batch file may be wrong.

Stack Errors

Numbers Left on the Stack

After compiling an application, always note the stack results. If there are numbers left on the stack, this always indicates an error. One common source for this kind of error is an incorrect use of the **[** and **]** compiler Words. Usually, an argument put on the GForth stack and has not been consumed during compilation of a MyForth Word.

One common error of this type is forgetting to use **#** or **##** after constants (literals) to be put on the MyForth stack at run time. Note that the action of these two Words is to take a number off of the GForth stack and compile a definition that will put the number on the Target's stack at run time (i.e., to compile a literal from a number placed on the GForth stack).

One way to see where errors of this kind occur is to look at the numbers left on the stack and then examine recently-edited code for those same numbers. If you have made a large number of changes to multiple files, then you can troubleshoot by putting unique numbers in between definitions to force them to appear on the stack display.

It is recommended that the troubleshooting numbers be put on lines by themselves, perhaps separated by blank lines. After recompiling, you can then see where the "bad" numbers appear in relation to the troubleshooting numbers.

One way to implement this technique is to use troubleshooting numbers that correspond to the line number in the source file. This also makes them easier to remove after the culprit is found. To isolate the problem down to a particular Include file, put line numbers before and after Include statements in the Job file.

Another way to troubleshoot this kind of problem is to place the following phrase in your code near the suspected problem definitions or "include" files, as follows:

```
[ cr .( before the bad words) .s ]
        .
        .
        .
[ cr .( after the bad words) .s ]
```

The troubleshooting phrases use GForth to display an error message and the GForth stack. These should help isolate the bad code.

Stack Underflow

If a stack underflow occurs during compilation, it is often caused by Words expecting an argument that is not provided. For example, the phrase "[push]" would cause an error because the left bracket has set the compiler up to search the GForth vocabulary first. This is common for assembly language sequences.

In this case, the "push" instruction requires a register number as an argument. The correct sequence would be something like: **[5 push]** . Note that the **push** and **pop** Words in MyForth are not assembler Words and do not require an argument.

One caution in using code from other applications is that the programmer may have used a Word knowing that the compiler state is correctly set or unimportant.

For example, "nop" will compile a "nop" instruction whether on not it is enclosed in brackets. Because it is an assembler Word, it is not defined in the MyForth vocabulary and it will be found when the GForth vocabulary is searched. We strongly encourage the explicit use of brackets to make the programmer's intent clear and avoid having to guess at the compiler state.

For example, the recommend phrase would be: **[nop]** . This explicitly specifies that this is an assembler Word and that the GForth vocabulary should be searched first (the GForth context is set by the first left bracket).

Numbers on the Target Stack

If you enter ".s" to look at the Target's stack and notice numbers that should not be there, check to see if you have loaded **debug.fs** .The debug file loads the definition for ".s" for the Target. If you have patched out debug.fs for the final application and forget to include it when going back to change or troubleshoot, then you will be executing ".s" for GForth, not the Target. Usually, you can tell if you are looking at the GForth stack if there are two stack items, both the number 4.

Locating Definitions

MyForth does not provide a facility such as “locate” (“see” in some systems) or “view” to locate the source code for a definition and automatically display or edit it. It also cannot display a list of all Words that use the definition (like the “where” command in some systems).

If you are using GVim you can locate the source for a Word by placing the cursor on the Word (e.g., in a definition) and then executing **Ctl]** (return with **Ctl T**).

Another work around for the lack of a “where” command is to use the “grep” command to locate a Word’s definition and all the places that it is used. This command is available on all Linux systems and can be installed with the Cygwin tools on Windows systems. Within GVim you can shell out (using the “sh” command) and execute grep from the command line.

For example, to find where the Word “test” is defined, enter: **grep “: test” –n *.fs** This will list all of the instances of the Word “test” and print out the file name, the line number (the “-n” option) and the contents of the line in which the Word appears.

This is also very useful in searching for possible duplicate definitions (another feature that MyForth does not have). If you cannot find a Word, remember that it may be defined as a macro (e.g., **grep “:m test” –n *.fs**).

Another way to find where a Word is defined is to compile the system (using the “c” or “d” commands) and then use “see” to observe what definitions are near the Word you are interested in. Often the names of the Words defined nearby will give a clue as to the name of the source file.

Serial Port

Hangup

By far, serial port hangup is the most common error you will encounter. This is typically because something in the application causes an error or because the application executes a non-terminating code sequence (e.g., begin … again). This is discussed more fully in the Downloading Problems section below.

Unfortunately, there is no way to recover from this error except by terminating the Command Window and restarting it. This is a reasonably fast operation if you have set up batch files that will quickly return you to your application directory.

Comm Errors

When compiling and downloading your application, you may notice an error that looks like this:

Problem downloading object code.
in file included from *the terminal*:0
***evaluated string*:-1: Aborted**
open-comm download target talking

This text indicates there is a serial port communications error. To recover, terminate the compilation (e.g., by executing **bye**), and then terminate the Command Window (e.g., by executing **exit**). After restarting the Command Window, the serial port should be reset and the application can be compiled and downloaded in the normal way (e.g., by executing the **d** command).

This error may also indicate that the serial port is not connected to or communicating with the processor. This could be because of a bad cable connection, lack of power or other problems that would keep the processor from communicating with the Host.

Downloading Problems

A number of downloading problems have been traced to USB serial adapters. All are not alike and some produce frequent connection problems during downloading. The best one found so far for Windows is an adaptor made by Keyspan. It is somewhat pricey but includes an LED indicator and software that can be used to monitor serial and USB traffic. Low-cost adapters have proven to be the most likely to cause problems.

One other very frustrating download problem is the inability to download after multiple tries, even with new command prompt windows. Turnkeyed applications performing high-speed I/O most frequently cause this type of problem. Applications performing serial port I/O can also cause a problem, particularly if they are turnkeyed.

The best way to prevent downloading problems is to avoid turnkeying an application until close to the time of final release. It is recommended that, during development, the "go" turnkey Word be executed manually after compiling and that the application loop have some way of manual termination (e.g., a switch press or power cycling).

One good way to produce an application that is very close to the final turnkeyed product but that can still be manually terminated is to test for a switch closure in the main application loop. Most of the Silicon Labs (SL) development boards include a utility pushbutton that can be used for this purpose. Try to arrange your I/O assignments so that this pushbutton remains mapped to the default port/pin assignment on the development board. Here is an example "go" sequence for a SL 310 development board:

```
:m sw  [ 7 .P0 ] m;  \ default assignment for on-board switch

: go   .version   init   begin  do-app  sw  0=until.  ;
```

Note that the loop terminates on a bit condition, hence the use of "0=until." (the period after "0=until" signifies that a bit condition is being tested, not the stack).

In the above example, the "sw" pin should be configured as an input. The switch pin is typically tied high through a resistor with the switch wired between the port pin and ground. The schematic for most of the SL development boards shows how this is done.

If all else fails, use the SL IDE to reprogram the chip using the debug adapter, as noted in the section titled “Development with the Debug Adapter” in the Tethered Target chapter.

Whenever a “c” or “d” command is executed, the application is recompiled and an Intel HEX file is generated as the **chip.hex** file (a code image in Intel HEX format). Specify this as the download file in the SL IDE. Be sure to use the **chip.hex** file in your application’s directory – the SL IDE will use the directory that was last used which is not necessarily the directory in which your application resides.

After programming the chip with this file and terminating the IDE, the application can be run after a power reset. Generally, disconnecting the debug adaptor is not necessary, but the chip should be disconnected before leaving the IDE.

Note that connecting to the chip, downloading the image file and disconnecting the chip are all performed from the “Debug” dropdown menu on the SL IDE.

If the revised application is set as “tethered” in the Job file with a switch termination in the main loop, as recommended above, the new program should respond reliably to download requests after the switch is pressed to terminate the “go” (turnkey) Word.

Using the SL IDE to download files will quickly convince you of the advantage of interactive downloading and development.

Improper Exits

The most frustrating type of error is caused by the unexpected consequences of a “return” compiled in a definition that is used within another definition.

Often this is caused by using a conditional macro within another conditional structure in a “colon” Word. An exit (;), defined in a macro, can cause an unexpected exit from the middle of a definition that uses the macro. Thus, it may “short circuit” the code following the macro. This is because the exit will be from the routine using the macro.

One symptom of this is that arguments are left on the Target’s stack after executing a “stack neutral” routine. To make matters worse, the number of arguments left on the stack can vary, depending on conditional execution within the definition.

Seemingly innocuous conditional phrases used within a definition are the most common cause of this error. Here is an example:

```
\ --- check for option change or switch closure

\ note: ?cmd can't be a macro because the first “;” exits ?change
: cmd?  ( -- flag)   sw  0=if. $ff # ; then $00 # ;

: ?change
   option dup  begin drop option |over xor cmd? ior until
   drop drop ;
```

In this example, **cmd?** is simply leaving a flag on the stack based on the condition of a switch attached to an I/O pin. (sw).

A problem occurs if **cmd?** is defined as a macro. Then, the return instruction compiled by the first “;” would cause a premature exit from **?change**, not an exit from **cmd?,** as intended.

With **cmd?** defined as a macro, the two **drop** instructions would not be executed and there could be another argument left from intermediate results within the **begin … until** loop.

Note that this “short circuit exit” can be very useful when correctly employed. If you are getting some very strange results of this type, the simplest solution is to closely examine your code and change some definitions from macros to Words.

Another (recommended) approach is to examine the code compiled by the troublesome definition and see under which condition the loop exits – you will usually find an unexpected (and unwanted) “ret” instruction. Remember, one of the primary programming activities in MyForth is examining compiled code using the **see** or **decode** instructions.

sees

As noted in the manual you can use the **sees.bat** file decompile a definition and write the output to a file using the “>” file redirection operator.

For example, to decompile 20 lines of the Word **2tib**, one could execute the following at the Command Window prompt: **sees 20 2tib >seeslist.txt** .

Unfortunately, **sees** cannot be used to view the decompilation of any Word containing the “>” character because Windows interprets it as a redirection operator.

The easiest workaround for this problem is to rename the Word (e.g., from **>tib** to **2tib**). Otherwise, you can use **see** or **decode** while at the MyForth prompt (perhaps using a screen print).

Conditionals

Most MyForth conditionals, such as **0=if** or **–if**, operate on the current value of the stack.

But, the following require a literal value supplied at compile time:

=if <if =until <until .

A compilation error will occur if these conditionals are used as if they are operating on stack values supplied at run time. Here is an example of a definition that correctly uses **<if**:

```
:m  |1char  ( n - n')  dup 48 # <if drop !err ; then drop adjust  m;
```

Also note that some conditionals leave the condition value on the stack and others, most notably conditionals based on bit tests, do not. One common error is the use of a “drop” with a conditional that tests a bit without affecting the stack.

One example would be the **if.** conditional used to check the status of a bit. One example of this would be the use of a phrase such as **4 .P0 if.** to check the state of a port bit.

The best way to avoid problems of this type is to examine working code examples. This is the kind of problem that will occur frequently when you first start coding but will rarely occur as you become familiar with how MyForth handles conditionals (e.g., at first, look at how the definitions using conditionals decompile).

Appendix A

Program Listings

```
\ 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
For application information:   http://www.amresearch.com

[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.
```

```
\ ----- 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;
```

```
: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.
```

```
\ ----- 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 $80 cond end 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 ----- /
: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;
:m <if =if then if' m; \ Is T <= literal?.
:m =until  =if end m;
:m <until  <if end m;
```

```
\ ----- More stack operations ----- /
:m # ?dup $74 , , m;      :m ## [ dup ] # [ 8 rshift ] # 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 u1+ $06 , m;    :m u1- $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;

\ ----- 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 $e5 , , 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;

\ ----- Definite loops ----- /
\ use, for example:
\    0 begin  dup . 1- 0=until drop
\ as a for next type loop.  The loop counter rides on top of the data
\ stack.

\ ----- Special cases ----- /
\ :m ADuC816 0 # $d7 #! m; \ Sets clock at 12.582912 Mhz.

0 org : reset

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

:m sees  ( n <word> - )   \ see n lines of word disassembly  11Aug07 rjn
        ‘ >body  [ @ ]  decodes 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 – Compiles and downloads the 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, one line at a time. To advance to the next line, use the space bar, "n", or any keys except the escape keys. To escape, use **q**, **Esc** or **Ctl-C**.
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
job.fs	Contains the definitions and files to be included to make up your application and also configures for your selected chip. This is the file compiled when you execute the **c** or **d** commands. Template files for this are in the chip directory (e.g., job300.fs or job120.fs).
main.fs	By convention, this is usually 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 binary for the compiled image of your application.
chip.hex	Contains an Intel Hex representation of **chip.bin**
tags.fs	**GForth** source code file to produce a tags file. The tags file contains the names of definitions and the names of the files that define them. Editors such as Vim or EMACS use the tags file to go to definitions from the text editor. For example, in Vim, placing the cursor on the word and pressing “CTL-]” will go to the definition under the cursor. For use with GVim (highly recommended), a reference to this file must be put in the GVim 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 (these take some patience to learn):

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.**

========== EDITING COMMANDS ==========
---------- Note: press Esc to exit edit mode ----------

a A	append text after cursor; end of line (starts insert mode)
<nn>dd	delete <nn> lines (dd - current line, use with p for cut/paste)
dw	delete the word after the cursor
D	delete the characters under the cursor to the end of line into x
d$	delete from cursor to the end of line
gf	goto file under cursor
i	insert text before cursor (starts insert mode)
I	insert text at the beginning of the line
J	Join the current line and the following line
O	open a new line below the cursor (starts insert mode)
p	place text from buffer (use with **y** to cut/paste) – ***very useful***
.	repeat last command, show file/directory list – ***very useful***
u	undo last change
x	delete character under cursor
yy	(yank) current line to buffer (use with p to cut/paste) – ***very useful***

========== COMMAND LINE ==========

:e	open file
:m	copy line
:x,ymz	copy line range x,y to just after line z – ***very useful***
:ls	show buffers
:w	save current file
:wq	write file and quit
:wqa	save and exit
:q	quit (append a **!** to discard edit buffer)
:qa!	quit without saving
:sav	save as
:s	substitute (e.g., :1,200s/hello/goodbye or :%s/hi/bye/gc)
:! <cmd>	execute the specified shell command (e.g., ":! dir")
ZZ	write current file to disk and exit (caps are important) – ***very useful***

========== SEARCH & NAVIGATION ==========

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

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

Ctl-]	Go to file & definition for the tag (Word, macro) under the cursor
Ctl-6	Go back to previous tag (Word, macro)
Gvim –t <tag>	Open file containing tag for editing

========== OPTIONS ==========
:set <option>

nu -OR- **number**	show line numbers – ***very useful***
nonu	no line numbers – ***very useful***
autoindent	indent to match previous line
ignorecase	ignore case on searches
shiftwidth=width	width of columns when using auto indent
tabstop=spaces	tab stop size
wrapscan	search from beginning of file when end is reached
nobackup	eliminates automatic backup creation – ***very useful***

Auto Commands:

au GUIEnter * simalt ~x opens in full window (alt space x)

To automatically detect file extensions, put something like this in the C:\Program Files\Vim\vimfiles\ftdetect file:

au BufRead,BufNewFile *.f set filetype-forth

========== 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
Ctl-Ws	split window
Shift-G	end of file
1G	start of file
4G	fourth page

========== 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

========== CONFIGURATION ==========
For Windows, put configuration information in C:\Program Files\Vim_gvimrc. For example, you can put various “set” and “auto” commands in this file per the OPTIONS section above.