430eForth for LaunchPad

Chen-Hanson Ting

Offete Enterprises, Inc.

2012

Chapter 1. eForth for LaunchPad

1.1 LaunchPad as a Firmware Development Platform

All these years, I have been looking for microcontroller platforms on which I can teach people how to program in the FORTH language. I designed a training course I called Firmware Engineering Workshop. I could train an open minded engineer to program in FORTH in about a week, with a reasonable capable platform, i.e., a microcontroller evaluation board with a FORTH operating system loaded. Good platforms are expansive, and low cost platforms are inadequate. What I did was to grab any microcontroller board at hand and used it. It did not work well because what I taught could not be easily replicated by people at home. People got frustrated when they could not reproduce results I demonstrated. Then, TI gave us the LaunchPad Kit.

The microcontroller evaluation board I need must have a microcontroller with reasonable capabilities. An 8-bit microcontroller with a fast clock is adequate. 16-bit of 32-bit microcontrollers are of course much better. The board must have at least 8 KB of ROM memory and 1 KB of RAM memory. It must also have a USART port to communicate with a terminal emulator on a host PC. Any other I/O devices will be icings on the cake. The more the better.

LaunchPad has all the components I listed above. It is also inexpensive, costing only $4.30, including shipping. It is a joke. I guess TI is desperate to compete against Auduino and Basic Stamps. It uses MSP430G2553, a very interesting 16-bit microcontroller which has 16 KB of flash memory, enough to host a FORTH operating system, 512 Bytes of RAM and many I/O devices to build substantial applications. LaunchPad Kit also has a USB port which connects a PC and an USART device in MSP430G2553. This serial interface is necessary for a FORTH system so that you can run and program MSP430G2553 interactively from a terminal emulator on the PC.

LaunchPad is a lovely kit. You connect it through a USB cable to your PC, and you can program it to do many interesting things. Its microcontroller MSP430G2553, running at 1.1 MHz, is very capable of doing many interesting applications.

It is a very nice platform to program MSP430G2553 in the FORTH language. FORTH exposes MSP430G2553 to you. You can interactively examine its RAM memory, its flash memory, and all the I/O devices surrounding the CPU. You can incrementally add small pieces of code, and test them exhaustively. An interactive programming and debugging environment greatly accelerates program development, and ensures the quality of the program.

The ealier version of LaunchPad used MSP430G2231 chip, which has only 2 KB of flash memory. It is too small to host a complete FORTH system. I built a 430uForth system for it, with only an interpreter and a small set of commands. It demonstrated that we could use the LaunchPad to do something interesting. Now that TI delivers it with MSP320G2553, we can have a complete FORTH system with

interepreter and compiler. It is a good platform for firmware engineering projects.

Since 1990, I have been promoting a simple FORTH language model called eForth. This model consists of a kernel of 30 some primitive FORTH commands which have to be implemented in machine instructions of a host microcontroller, and 190 compound FORTH commands constructed from the primitive commands and other compound commands. By isolating machine dependent commands from machine independent commands, the eForth model can be easily ported to many different microcontrollers. This model is ported to MSP430G2553, and the result is the 430eForth system, which runs very nicely on LaunchPad Kit.

430eForth is written in MSP430 assembly. The code is provided so that you can modify it to suite your application. The entire system takes up about 6000 bytes of the flash memory, leaving lots of room for your application.

Needless to say, the heart of an LaunchPad is the MSP430G2553 microcontroller. If you like to fully understand LaunchPad and make the best use of it, eventually you have to deal with MSP430G2553 directly. You will have to come back and read the "MSP430x2xx Family User's Guide" (slau144) from TI Corp, which is a huge 658 page document. It is a dry technical document, not for casual reading. Actually, it is not that bad. Only when you have to drive one of the devices, like the I/O devices, the control and status registers, etc., in MSP430G2553, you open the respective chapter and learn all about this device, line by line, word by word. If you have 430eForth running, you can examine the associated registers, and all the bits in these registers will gradually make sense. Change these bits interactively, and observe the effects. There is no better way to learn these devices, and to make them work the way you want them to work. And, 430eForth is your best friend to do that.

1.2 What is FORTH?

FORTH was invented by Chuck Moore in the 1960s as a programming language. Chuck was not impressed by programming languages, operating systems, and computer hardware of that time. He sought the simplest and most efficient way to control his computers. He used FORTH to program every computer in his sight. And then, he found that he could design better computers, because FORTH is much more than just a programming language; it is an excellent computer architecture.

So what is FORTH really?

Many books and many papers had been written about FORTH. However, FORTH is still elusive because it has many features and characteristics which are difficult to describe. Now that it has moved from software to hardware, with technologies like FPGA and custom IC, it is even more difficult to accurately put it into words. Here I will try to look at it from a completely different angle.

FORTH is a list processor. It is very similar to LISP in spirit, but totally different in form. Both languages assume that all computable problems can be expressed and solved in nested lists.

FORTH has a set of commands, and an interpreter to process lists of commands.

FORTH commands are records stored in a memory area called a dictionary.

A record of a FORTH command has three fields: a link field linking commands to form a searchable list, a name field containing the name of this command as an ASCII string which can be searched, and a code field containing executable code and data to perform a specific function for this command. It may have an optional parameter field, which contains additional data needed by this command. The link field and name field allow the interpreter to look up a command in the dictionary, and the code field provides executable code to perform the function assigned to this command.

A FORTH command has two representations: an external representation in the form of a text string with ASCII characters; and an internal representation in the form of a token, which invokes executable code stored in a code field. In many FORTH systems, the tokens are addresses. However, tokens can take other forms depending on implementation. For example, Java, which is a variant of FORTH, uses byte tokens.

There are two types of FORTH commands: primitive FORTH commands having machine code in their code fields, and compound FORTH commands having token lists in their code fields.

The FORTH interpreter processes lists of commands in text strings. A list of FORTH commands contains a sequence of strings representing FORTH commands, separated by white spaces and terminated by a carriage return. The interpreter parses out commands in the text strings into tokens and executes code represented by these tokens. When the FORTH interpreter encounters a primitive command, it executes the machine code in its code field. When it encounters a compound command, it processes the token list in its code field. How it processes the token list depends upon how tokens are defined and implemented.

The text interpreter operates in two modes: interpreting mode and compiling mode. In the interpreting mode, a list of command names is interpreted; i.e., commands are parsed and executed. In the compiling mode, a list of command names is compiled; i.e., commands are parsed and corresponding tokens are compiled into a token list. This token list is given a name to form a new compound command, adding a new command record in the dictionary.

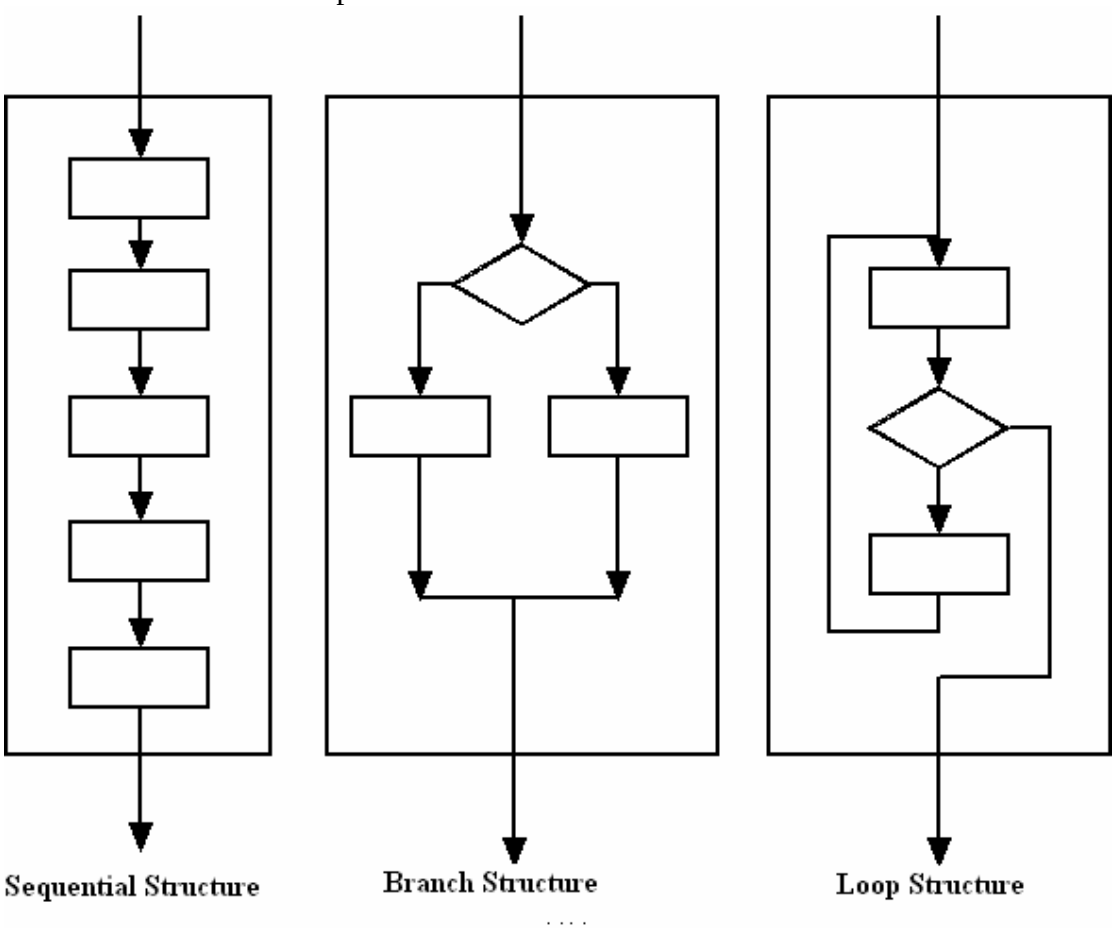
New compound commands are compiled to represent new token lists. This is the most powerful feature of FORTH, in that you can compile new compound commands, which replace lists of existing commands, both primitive and compound. The syntax to compile a new compound command is:

```
: <name> <list of existing commands> ;
```

Nested token lists are added as new compound commands until the final compound command becomes the solution of your problem. Lists are compiled and tested from the bottom up. The solution space can be explored wider and farther, and an optimized solution can be found more quickly.

Linear, sequential token lists are enhanced by control structures like branch structures

and loop structures. A structure is a token list inside which the execution sequence can be modified dynamically. The following figure shows a sequential structure, a branch structure and a loop structure.



A structure has only one entry point and one exit point, although it may have many branches inside. Structures can be nested, but may not overlap with one another. A structure can therefore be considered an enhanced token. A compound command is a structure given a name.

Using the concept of structures, a new compound command has the following syntax:

```
: <name> <list of structures> ;
```

The fundamental reason why FORTH lists (command lists and token lists) can be simple, linear sequences of commands is that FORTH uses two stacks: a return stack to stored nested return addresses, and a parameter stack to pass parameters among nested commands. Parameters are passed implicitly on the parameter stack, and do not have to be explicitly invoked. Therefore, FORTH commands can be interpreted in a linear sequence, and tokens can be stored in simple, linear token lists. Language syntax is greatly simplified, internal representation of code is greatly simplified, and execution speed is greatly increased.

A FORTH Virtual Machine thus needs two stacks, efficient means to traverse nested token lists, and a CPU within a reasonable instruction set and memory device to support a small number of primitive commands. eForth is such an implementation which has been ported to many commercial microprocessors and microcontrollers.

Auduino Kit with an MSP430G2553 microcontroller, is an ideal platform for an eForth implementation, 430eForth system.

1.3 FORTH for Firmware Development

To use FORTH to develop applications for MSP430G2553 with LaunchPad Kit, you have to have the following components:

First, you need the $4.30 LaunchPad Kit with an USB cable connecting to PC. Second, on the PC, you need Code Composer Studio 5.2, an Integrated Development Environment (IDE) from TI Corp to assemble 430eForth. You can download it for free from www.ti.com.

Code Composer Studio 5.2 contains an MSP430 assembler, C and C++ compilers, and a debugger. It also loads assembled or compiled object code to MSP430G2553 through the USB cable. I only use the MSP430 assembler to assemble the source code of 430eForth. Once 430eForth is loaded to MSP430G2553, all programming and debugging operations are performed from a terminal emulator on PC, through the USB cable connected to LaunchPad Kit. USB drivers are installed automatically when you install Code Composer Studio.

On the PC, I use HyperTerminal to communicate with LaunchPad Kit. HyperTerminal comes with Windows, and can be accessed through \Start\All Programs\Accessories\Communication\HyperTerminal. Starting at Windows 7, Microsoft stopped bundling HyperTerminal with Windows. However, you can still download HyperTerminal application from MSDN website.

There are other terminal emulators for PC to communication with LaunchPad. RealTerm can be downloaded from SourceForge (http://realterm.sourceforge.net/). It has many more options than HyperTerminal, but they work similarly.

You have to set up communication protocols on HyperTerminal or RealTerm so that they will communication with LaunchPad. The set up parameters are 2400 baud, 1 start bit, 8 data bits, no parity, 1 stop bit, and no flow control.

To develop programs for embedded systems, the conventional methodology is to write source code in C or in assembly. The source code is compiled or assembled. Object code is linked by a linker to produce execution code, which is loaded to the target system. Now, you cross your fingers and turn on power. Most likely, the system does not work, and you enter into the debugging phase of development.

To debug a program in an embedded system, you need lots of sophisticated tools, like simulator, in-circuit emulator (ICE), an oscilloscope, and a good logic analyzer. You set up break points, and trace the microcontroller instructions cycle by cycle. It is very difficult when the application program is large and complicated, especially when you can only observe the microcontroller from the outside.

The Code Composer Studio streamlines the programming process. You write your code in its edit perspective. You press the compile button to compile the assembley code. Then, you cross your fingers and press the load/run button. If it works, great

for you. If it does not work, you can get lots of help in the debugger perspective. You can set up break points, You can single step through the code. You can watch memory, registers and IO devices. Debugging is not an easy job, even with Code Composer Studio.

FORTH provides you the proper tools. You embed the debugging tools inside the microcontroller in the form of an interactive FORTH operating system. Source code in the form of many small commands is compiled by the target microcontroller in the embedded system. You can control the microcontroller from within, and observe its behavior from inside out. Break points are not necessary, because FORTH commands naturally break at their ends, and you can query their results interactively. New commands are compiled, tested, and debugged incrementally. The solution space can be explored quickly, and almost exhaustively. Reliable system can thus be built quickly. FORTH commands are lists of nested lists, and are very compact. Substantial applications can be stored in very small memory area.

Chapter 2. 430eForth for MSP430

2.1 Introduction

For a very long time, firmware engineering meant to program a UV Erasable PROM chip and to insert it on a board which contained a microcontroller, some RAM memory chips, and some I/O chips, and a socket for the UV EPROM. Then flash memory chips replace UV EPROM's. And then everything is integrated into a single microcontroller chip, and we now have ISP, In System Programming, which allows you to program the microcontroller in its own socket. LaunchPad Kit integrates an MSP430G2553 microcontroller with all necessary hardware components on a small printed circuit board, and captures the fancy of a new generation of will-be firmware engineers and DIY hobbyists. After 20 years of implementing eForth on many different microcontrollers, I am certainly of the opinion that eForth is the FORTH best suited for microcontroller.

The original eForth was implemented in Direct Thread Model by myself and Bill Muench. Dr. Richard Haskell implemented the first Subroutine Thread Model in 86se4th.asm for 8086 and 68000. I took the original eforth86.asm file and modified it so it could be assembled by the MSP430 assembler in Code Composer Studio 5.2 development system from TI. I call it 430eForth because it is configured specifically for MSP430G2553, used on LaunchPad Kit.

The most important features of 430eForth are the following:

1. Subroutine Thread Model.
2. Using byte addresses to access flash and RAM memory.
3. All assembly code are in a single 430eForth.asm file.
4. New FORTH code are written directly to flash memory.
5. No interrupts and no multitasking.
6. Information flash memory Segment D is used to initialize variables.
7. Ease in building turnkey applications

These features make 430eForth a very simple, easy to use, easy to understand and easy to modify. That why FORTH is prefixed with an "e".

2.2 Installing Tools

Here are the steps you can follow to get everything running.

Get an LaunchPad Kit board from DigiKey for about $4.30.

Download the Code Composer Studio 5.2 from TI web site:
http://www.ti.com

Install Code Composer Studio 5.2. Do not connect the USB cable until the software installation is complete.

To check on these USB drivers, plug in the cable and go to Start\Control-Panel\System\Hardware\Device-Manager. Under Ports (Com & LPT),

you will see MSP430 Application UART(COM X). Remember the COM port number X for use with HyperTerminal or RealTerm.

2.3 Assembling 430eForth

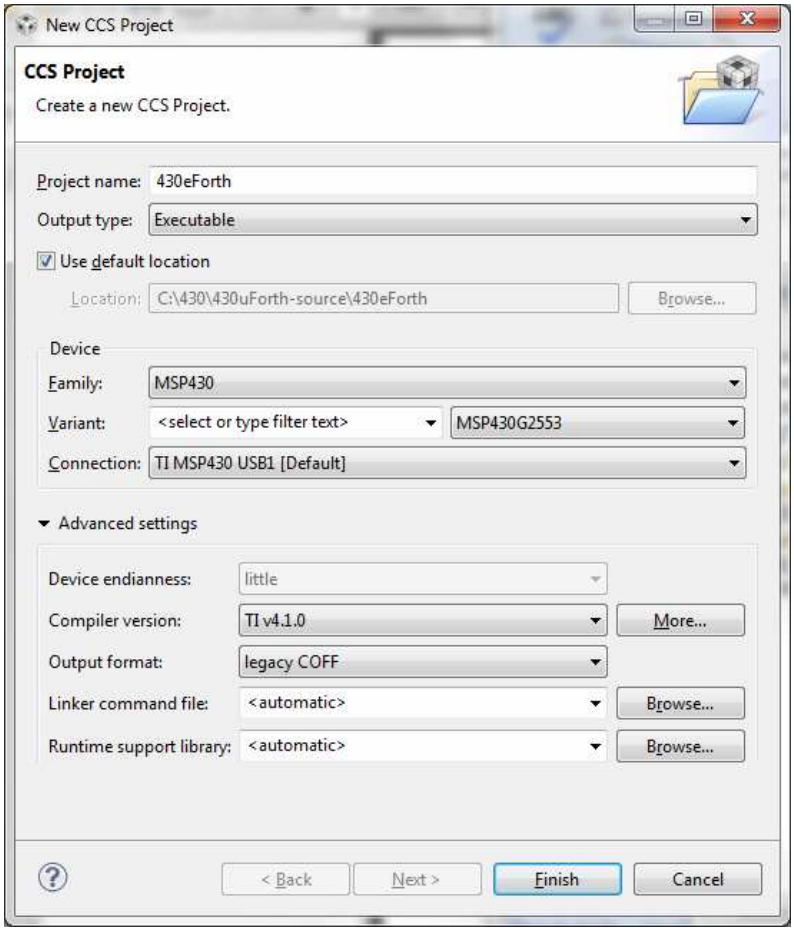
You have to be thoroughly familiar with Code Composer Studio 5.2 in order to get it assembling 430eForth and get the LaunchPad to work. Follow the two CCS documents Slau157 (Code Composer Studio v5.1 User's Guide for MSP430) and Spru509 (Code Composer Studio Development Tools v3.3 Getting Started Guide). I will not repeat the steps that you must go through to get CCS up. I will only highlight the steps that are essential to get the 430eForth system assembled and running.

Code Composer Studio presents its window in "Perspectives". A perspective is a collection of panels showing relevant information about the project at certain stage of program development. The first perspective is the "Edit Perspective", which contains a Project Navigation Panel to the left, and text editing panel to the upper right, and Console Panel at lower left, and a Problem Panel at lower right. This is where you enter source code, do your editing, and assemble your code.

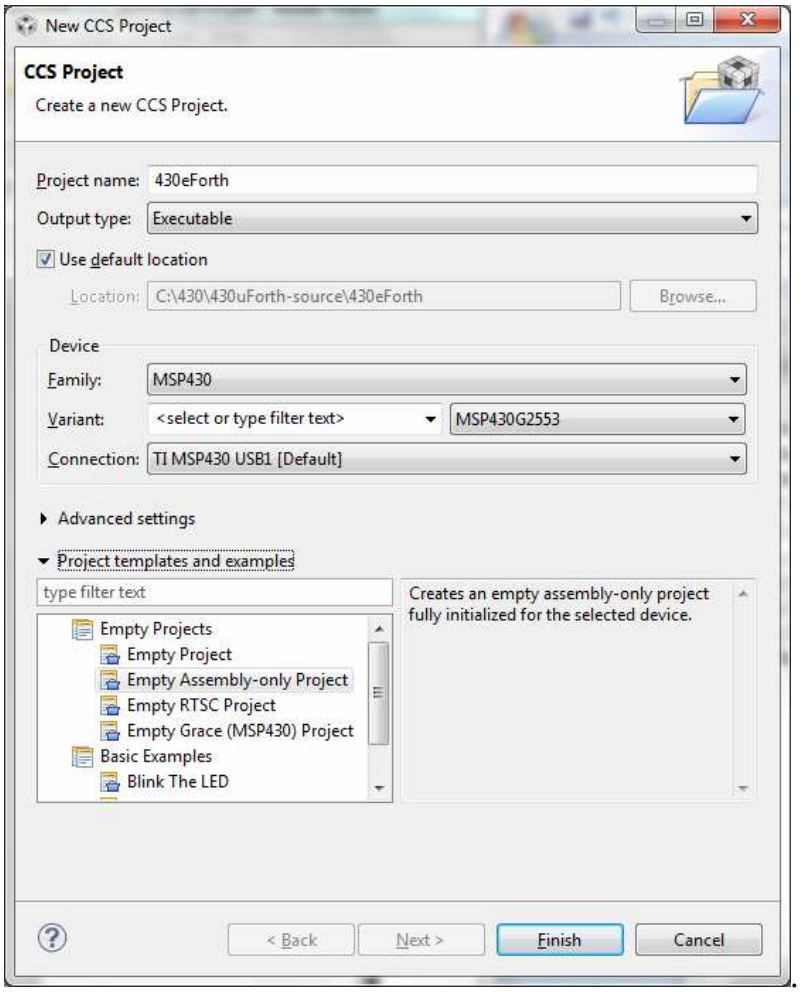
In the CCS window, select Project>New CCS Project. In the New CCS Project window, enter a project name, like 430eforth, in the Project Name box. A default path is shown in the Location panel. You can change this path by clicking the box to the right of Location panel, and then navigate to the folder you want.

Select MSP430G2553 as the Device.

In the Advanced Setting Options, change Output Format from ELF to COEF. This is very important. If the assembler sends out an ELF file, the linker will not recognize it and produces a fatal error. No .out file will be produced and you will be stuck in a deep hole. At this point the New CCS Project window looks like the following:

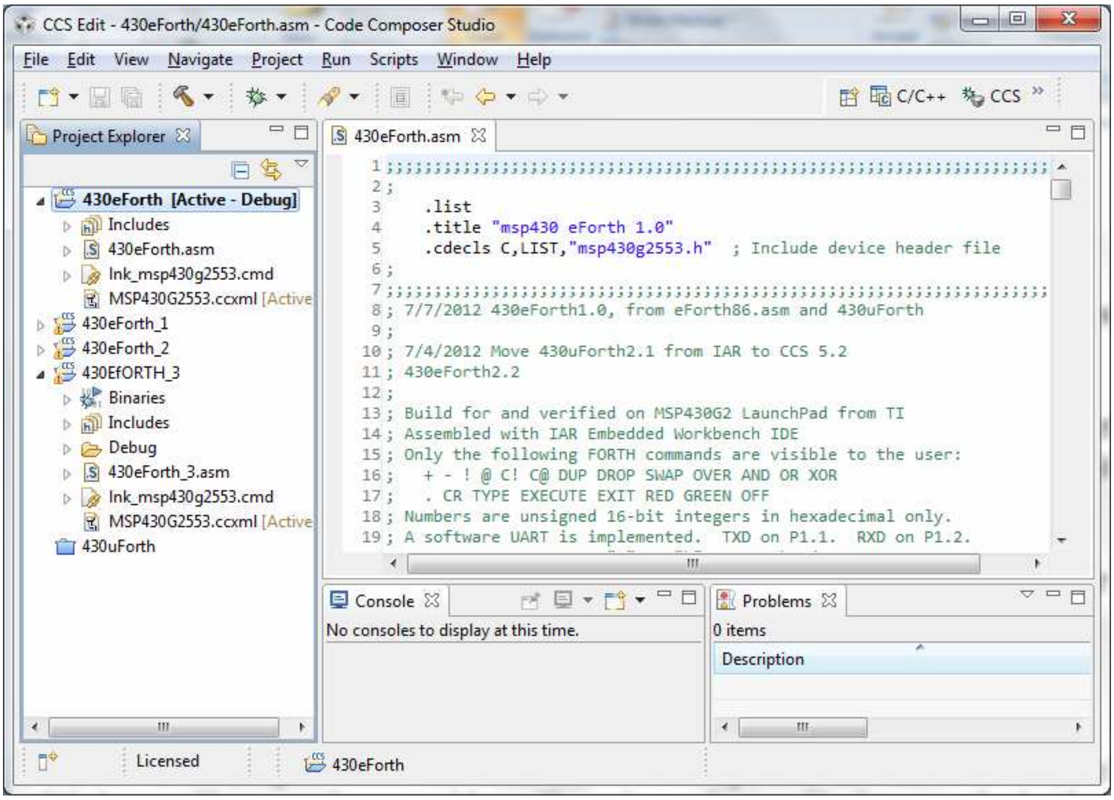


In the Project Template Options and Examples panel, select Empty Projects>Empty Assembly-Only Project option and the New CCS Project window looks like this:



Click Finish button and the Studio 5.2 Window shows you the new project. You are ready to go to work.

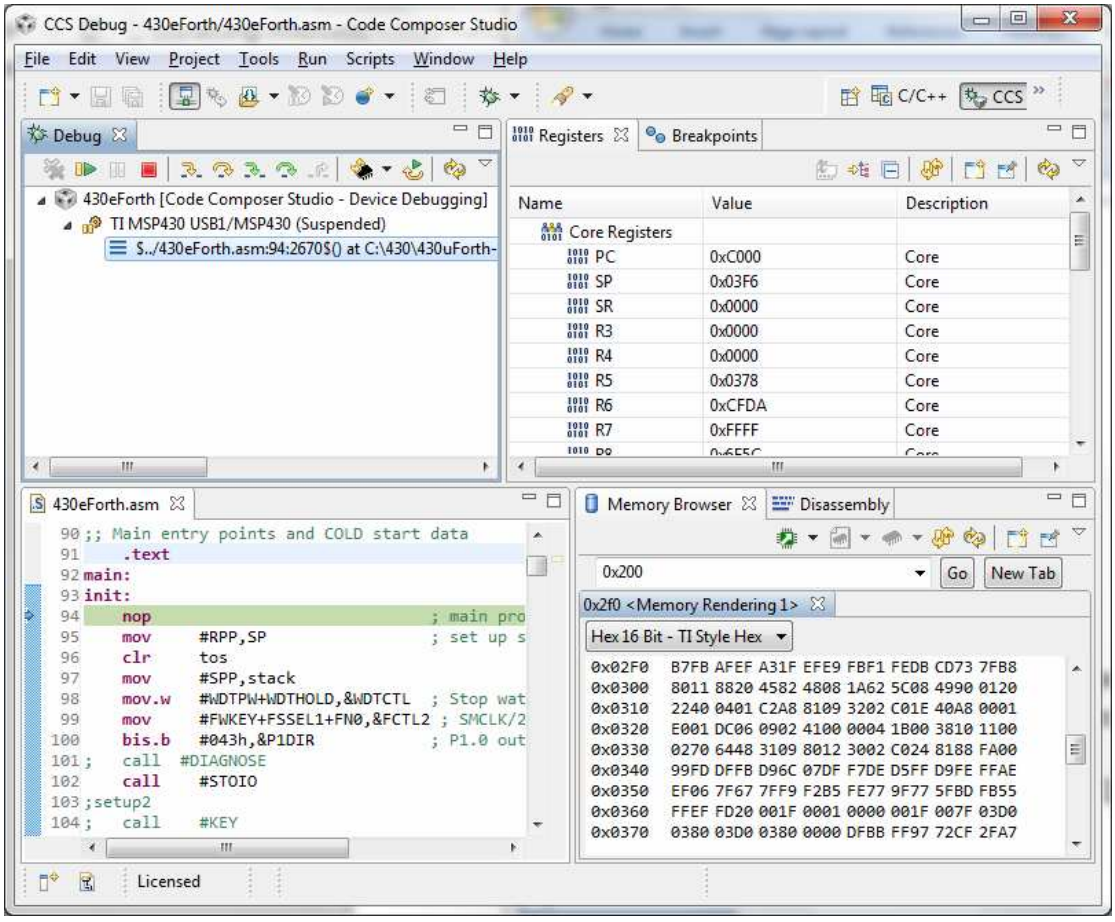
In the workspace folder CCS built for you, you will file the new project 430eForth as a new folder. Copy 430eForth.asm into this folder, and open this file in the Edit perspective. The CCS window appears like the following:



Pull down the Project Menu and select the Build All option. CCS starts assembling 430eforth.asm, and displays lots of messages in the Console panel at the bottom of the Edit panel. Its final message is: "Build Fiinshed”. Scroll up the Console panel, and you will see the most important message:
'Finished building target: 430eForth.out'

Assembling is successful. However, above it are 13 warnings. In the Problems panel to the right, it also shows the results: “0 error, 13 warnings, 0 others”. The linker does not find many of the interrupt vectors and is not happy. Ignore the warnings. If you are curious, you can Google the text “warning# 10374-D”, and find out its meaning. It there are error messages, you will have to correct the mistakes until the linker produces and .out file.

Pull down Run menu and select Debug option. You are now presented with a Debug Perspective, where you can test, debug and run 430eForth. In the following figure I show you my favorite perspective panels. The Debug panel is at upper left. The Registers/Breakpoints panel is at upper right. The Edit panel is at lower left. The Memory/Disassembly is at lower right.



In the Debug panel, the tool bar contains 12 buttons, since I cannot draw the graphs, I just list the buttons and show you what they do:

1. Remove all terminated launches
2. Resume
3. Suspend
4. Terminate
5. Step Into
6. Step Over
7. Assembly Step Into
8. Assembly Step Over
9. Step Return
10. Reset
11. Restart
12. Refresh

I mostly use Resume to start running, Suspend to stop running, and Terminate to stop debugging and return to the Edit Perspective.

When debugging, I use Assembly Step Into and Assembly Step Over. In the Registers panel, I always display registers R0 to R6, as R4 is TOS(Top of parameter stack) and R5 is the parameter stack pointer. In the Memory panel, I generally display RAM memory from 200H to 3FFH. The return stack is from 3F8H down, and the parameter stack is from 378H down. Watching the parameter stack generally

allows me to find problems and ways to correct them.

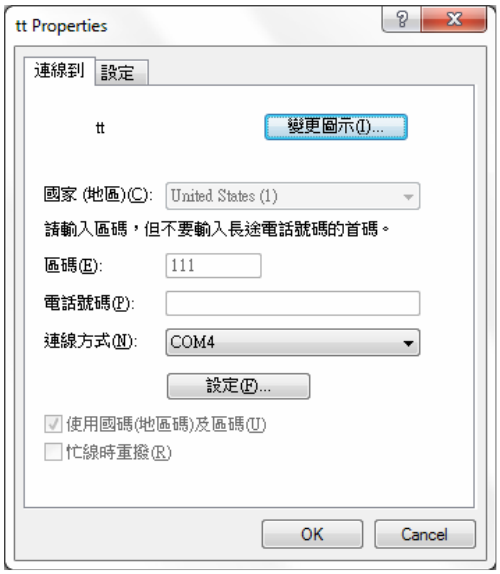
OK. In the Edit panel, the line of code after MAIN is highlighted, showing the instruction about to be executed. You can push the Step Into or Step Over buttons to step through the code. As 430eForth is fairly well debugged, you can push the Resume button to run it.

2.4 The Terminal Interface

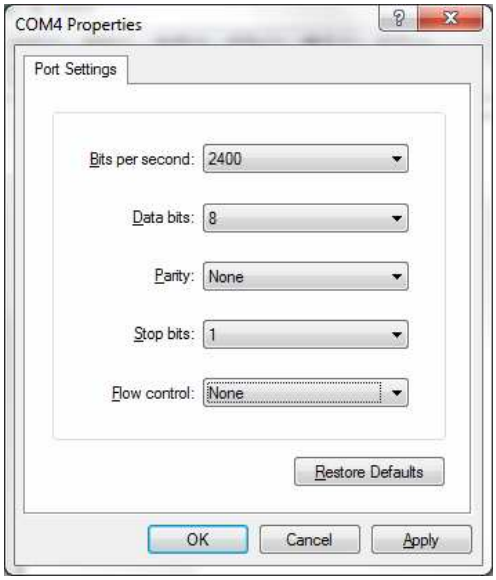
I forgot to mention that the LaunchPad Kit must be plugged in to the PC through the USB cable, and that you will have to have HyperTerminal started. HyperTerminal is bundled in Windows until Windows 7. If you are using Windows 8, Google it and find how to install it.

On the HyperTerminal console pull down the Call menu and select Disconnect option. Then, pull down the File menu and select Properties option. In the Connect Using dialog box, select the COM port you saw earlier in the USB device assignment.

On my PC, the HyperTerminal in in Chinese, and I have not learnt how to turn it back into English. You have to bear with me showing you the HyperTerminal windows in Chinese. However, I hope you are familiar with HyperTerminal and know what I am talking about.



Click the Configuration button and a COMx Properties window pops up. Select 2400 baud, 8 data bits, no parity, 1 stop bit, and no flow control. Then click OK button to dismiss the COMx Properties window.

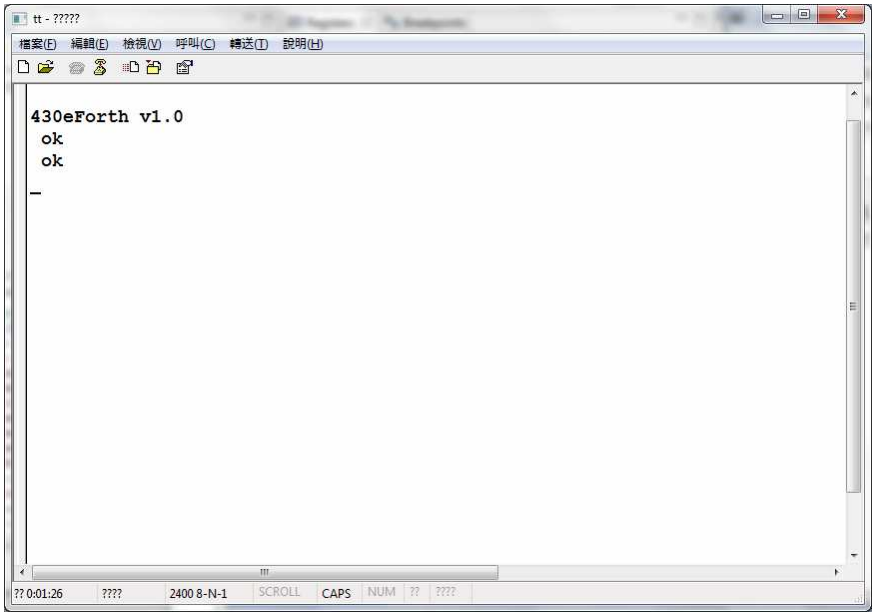


In the main Properties window, click on the Settings tab and the click the ASCII Setup button, and an ASCII Setup window pops up. Enter 900 in the Line Delay dialog box to insert 900 msec delay after sending each line of text. Later you will download source code files and you will need this end of line delay.

Click OK button to dismiss the ASCII Setup window. Click OK button in the main Properties window and dismiss this window also.

When HyperTerminal is set up, and the Resume button is pushed in CCS Debugger , both the red LED and green LED on the LaunchPad Kit lit up. 430eForth enters into a waiting loop for a “B” key. Not, hit the “B” key on your keyboard. Receiving a “B” key, 430eForth determines the baud rate, set up the software UART and enters into the text interpreter of FORTH. You will see the sign-on message generated by 430eForth:

```
430eForth v1.0
```



Now you can type in FORTH commands and 430eForth will execute them.

430eForth, like the original eForth Model, is case sensitive. Most of the FORTH commands are in the upper case. So, you probably want to push Caps Lock key to look the HyperTerminal in upper case mode.

Hitting Return key several times, and you should see `ok` messages are displayed on the HyperTerminal console. You can now type in FORTH commands to interact with 430eForth on LaunchPad Kit.

2.5 Testing 430eForth on LaunchPad Kit

To recapitulate, you have to install Code Composer Studio 5.2. You have to connect your LaunchPad Kit board to a USB port on your PC. Assemble 430eForth.asm, and download its 430eForth.out file to LaunchPad Kit. Open HyperTerminal on your Windows and you get the sign-on message:

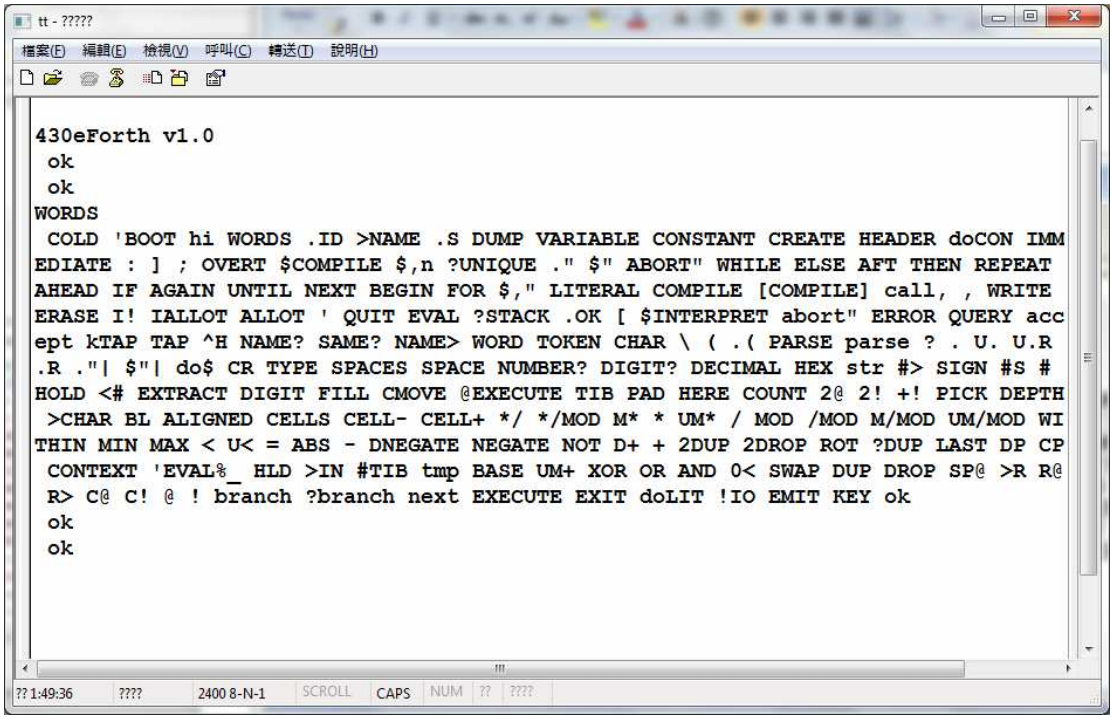
```
430eForth v1.10
```

Type these FORTH commands to test the system:

```
WORDS
HEX
200 DUMP
C000 DUMP
D700 DUMP
```

Note that 32eForth is in the hexadecimal base.

After bring up 430eForth, type `WORDS` and you will see a list of eForth commands on the HyperTerminal console:



HyperTerminal breaks up a word at the right margin of the window console. You will have to read across lines to see whole words. There are about 200 FORTH commands visible in 430eForth system.

These eForth commands are documented in the Appendix for your reference.

Make sure that HyperTerminal inserts a 900 ms delay after sending each line of text. Then, you can download a text file by pulling down Transfer Menu and select Send Text File option. From the file selection window, select a file and push the Open button. Or, double clicking the selected file. Text from the selected file will be sent to 430eForth, one line at a time, and you will see how 430eForth responds to these lines.

2.6 Learning More about eForth

If you are new to the FORTH programming language, or has some prior knowledge on a different FORTH system, you may want to look into a series of tutorials I prepared for the earlier eForth systems. There are 17 lessons in that many text files. Your are encourage to take these lessons and type in the commands. You can also download these files in HyperTerminal, and then type in the final commands to test loaded applications. These lessonXX.txt files are included in the distribution package with 430eForth.asm.

The contents of these lesson files are listed in the following table:

Lesson	Contents
1	Hello, World!
2.	Big characters
3.	Forth Interest Group
4.	Repeated patterns

5	The theory that Jack built
6	Help
7	Money exchange
8	Temperature conversion
9	Weather reporting
10	Multiplication table
11	Calendars
12	Sines and cosines
13	Square roots
14	Number conversion
15	ASCII character table
16	Random numbers
17	Guess a number

Chapter 3. Features in 430eForth Implementation

3.1 Memory Map

There are 16 Kbytes of flash main memory, and 512 bytes of RAM in MSP430G2553. In addition it has 256 bytes of flash information memory. These memories, CPU registers, and IO device registers are arranged as show in the following table:

Start Address	End Address	Name and Function
0	0FFH	Special Function Registers
10H	0FFH	8-Bit peripheral registers
100H	1FFH	16-Bit peripheral registers
200H	21FH	RAM, system variables
220H	--	RAM, free space
--	378H	RAM, parameter stack
380H	--	RAM, Terminal Input Buffer
--	3F8H	RAM, return stack
3F9H	3FFH	RAM, free space
1000H	103FH	Segment D, flash information memory
1040H	107FH	Segment C, flash information memory
1080H	10BFH	Segment B, flash information memory
10C0H	10FFH	Segment A, flash information memory
0C000H	0FFDFH	Flash main memory
0FFE0H	0FFFFH	Reset and interrupt vectors

Two pointers are used by eForth to manage the RAM and flash memories. CP points to the top of the dictionary In the flash main memory. When new commands are compiled, DP is increased to make room for new code and data. DP points to the top of the free space in RAM. When new variables and arrays are defined, DP is increase to allocate space in RAM.

Currently, the eForth system occupies flash memory from 0C00H to 0D788H. About 10 Kbytes are available for you to add new FORTH commands.

Initial values of system variables are stored in Segment D of the flash information memory. This segment can be erased independently from the flash main memory. When you are satisfied with the application you have developed, erase Segment D and copy the current values of variables into it. When MSP430G2553 chip is reset, or when the LaunchPad is powered up, your application will run immediately. This is how you build turnkey systems on the LaunchPad.

3.2 Flash Programming

MSP430G2553, with its flash memory, is very friendly to FORTH. When 430eForth is downloaded from CCS to LaunchPad, the flash memory above the eForth dictionary is all erased, and new commands and data can be written into the flash memory with the eForth command `I!.`
From I!, a set for commands are defined to make it possible to compile new FORTH

commands. These commands are shown in the following table:

Command	Stack Effects	Function
I!	n a --	Write data n into flash memory at address a.
,	n --	Compile data n to the top of dictionary. CP is incremented by 2. It is the primitive FORTH compiler.
ERASE	a --	Erase one page of flash memory. One page is 512 bytes for flash main memory, and 64 bytes for flash information memory.
WRITE	src dest n --	Copy n bytes from src to dest. Dest must be an address to the flash memory.

When you compile new words, they are added to flash, but there is no easy way to "forget" them. The flash must be erased in 512 byte pages, and it is difficult to compile words in independently erasable pages. We do not have enough RAM memory to store a page of code, erase this page in flash, make changes in RAM, and write the new page back into flash memory.

This is the way to do code development:

1. Compile and test you code. Redefine the code repeatedly until flash is full.
2. Reload 430eForth, and flash is erased. Compile verified code first. Then go to Step 1. Compile and test new code. And so forth.
3. When an application is done, load the application into a fresh 430eForth system.
4. Erase Information Flash Segment D by: `HEX 1000 ERASE`
5. Copy system variables back to Segment D by: `200 1000 20 WRITE`

Now you have a turnkey application, which will boot up when 430 is reset or power-up.

3.3 Software UART

MSP430G2553 on the LaunchPad does not have an external clock. It runs on the DCOCLK, internal digitally controlled oscillator, at 1.1 MHz. This clock is not accurate enough to generate baud rate clocks for a UART. 430eForth therefore includes a software UART which can lock to an external UART device by detecting a “B” character from the external UART. When 430eForth boots up, if falls into a waiting loop for the “B” character. With the “B” character, it calculates the baud rate of the external UART and uses this baud rate to transmit and receive characters.

Although MSP430G2553 does have a hardware UART device, it uses two TX/RX pins incompatible with the USB interface on the LaunchPad Kit. For compatibility reasons, I keep the software UART. The limitation is that it runs well at 2400 baud. It becomes unstable at higher baud rates.

3.4 Files

MSP430G2553 has only 512 bytes of RAM memory, and it is not enough to handle files and other mass storage requirements. At present source files are sent to 430eForth for compiling through the serial terminal USB/COM port. To allow for interpretation and compilation, a pause must be inserted at the end of each line of text

sent to 430eForth. I set the end of line delay in HyperTerminal to 900 ms. It probably could be half this value. Upon a compiling error an error message will be shown, but execution continues as the next lines of text are still streaming out of the serial port. You must manually watch for compilation errors. Generally, one error will cause many other errors, and 430eForth would crash if it encounters serious errors. When this happens, reload 430eForth from CCS.

3.5 Case Sensitivity

eForth is case sensitive, and must of its commands are in the upper case. It is possible to make it case insensitive, like what I did in 328eForth for Auduino. Let's see if there is a demand.

3.6 What 430eForth Does Not Have

430eForth has no compiler security to check on the pairing of conditionals when compiling structures. Having an extra `THEN` in a colon definition will almost certainly crash the system. In this case, execution will show odd errors; and you have to reload the 430eForth hex images. Do be careful when writing these structures:

```
IF…THEN
IF…ELSE…THEN
BEGIN…AGAIN
BEGIN…UNTIL
BEGIN…WHILE…REPEAT
FOR…NEXT
FOR…AFT…THEN…NEXT
```

Remember: Structures can be nested but cannot overlap.

430eForth does not support interrupts, multitasking, user variables, and local variables.

All commands in the 430eForth dictionary are linked in a single vocabulary. No multiple vocabularies.

430eForth does not have an assembler. If you have to code assembly routines, use the MSP430 assembler in Code Composer Studio 5.2.

All these features can be added to 430eForth. But, it is better to keep it simple so people can understand if fully. If you have specific needs for specific tasks, I am sure you can somehow implement them or have people to help you.

MSP430G2553 is a small microcontroller. 430eForth is a seed we plant in it. You can make it to grow into something useful for you.

Chapter 4. 430eForth Source Code

MSP430G2553 is a very interesting microcontroller from TI Corp. It has a 16 bit CPU with 16 registers, 16 KB of flash memory, 512 bytes of RAM memory, 256 bytes of flash information memory, and a host of I/O devices. It is produced in a 16 pin DIP package, with 14 I/O pins. It is ideally suitable for many embedded applications. Being a 16-bit CPU, it is a very nice host for a FORTH Virtual Machine.

In 430eForth system, we adopt the Subroutine Threading Model, in which command tokens are represented by subroutine call instructions, and a compound command consists of a list of subroutine call instructions. Nested token lists, as nested subroutine lists, are executed naturally by MSP430G2553 CPU with very little overhead in the nesting and un-nesting of subroutine calls and returns. It is also possible to mix tokens with CPU machine instructions when optimizing FORTH commands.

Using the Subroutine Threading Model, physically the compound commands has the identical structure as the primitive commands, and both types of commands are generally terminated by a ret machine instruction.

The CPU stack pointer register sp is used as the return stack pointer in the FORTH Virtual Machine, and the register r5 is used as the parameter stack pointer. Both the return stack and the parameter stack are located in the high end of the RAM memory area. The top element of the parameter stack is cached in register r4, called tos, and it significantly increases the speed in accessing the parameter stack.

Besides the stacks, the RAM memory area also contains 12 system variable, the terminal input buffer, a word buffer to parse input strings, and a text buffer to build numeric strings for output.

In the original eForth Model, only 30 primitive commands were defined to enhance its portability to a wide range of microcontrollers. In the 430eForth implementation, to make it run faster, many compound commands are re-written in MSP430 assembly code.

In the following sections, I will present the 430eForth system in its complete source listing. The source code is commented liberally. However, in-line comments are only adequate to document the functions of the source code, but not sufficient for the intentions behind the source code. To give myself enough room to discuss the structures and the design requirements of all the commands, for one section of source code, I add another section for comments. I hope this format will let me explain more fully what the commands do and what was intended for them to do.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;
;
    .list
    .title "msp430 eForth 1.0"
    .cdecls C,LIST,"msp430g2553.h"  ; Include device header
file
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;
; 7/7/2012 430eForth1.0, from eForth86.asm and 430uForth
;
; 7/4/2012 Move 430uForth2.1 from IAR to CCS 5.2
; 430eForth2.2
;
; 4/21/2011  430uForth
; Build for and verified on MSP430G2 LaunchPad from TI
; Assembled with IAR Embedded Workbench IDE
; Only the following FORTH commands are visible to the user:
;   + - ! @ C! C@ DUP DROP SWAP OVER AND OR XOR
;   . CR TYPE EXECUTE EXIT RED GREEN OFF
; Numbers are unsigned 16-bit integers in hexadecimal only.
; A software UART is implemented.  TXD on P1.1.  RXD on P1.2.
;   On power-up, press "B" or "b" to set baud rate.
;   Set terminal baud rate to 2400 baud.  Not stable at higher
rates.
;   Do not disturb TXD and RXD, else the UART will not talk.
;
; Try:
;   RED       turn on red LED
;   GREEN     turn on green LED
;   OFF       turn off both LEDs
;   20 C@     read P1 inputs.  Press S2 switch to see the
effects.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;
;
```

```
; Subroutine Thread Model of eForth
; Only the interpreterr is implemented due to memory
limitation.
; Return stack pointer is SP, TOS is R4, and data stack pointer
is R5.
; Variables TEMP, CONTEXT, #TIB, >IN and DP are in CPU registers
; R14 and R15 are used by the software UART, for baud rate
control.
; It works on MSP430G2231, but may work on other 430 chips.
; The only peripheral used in P1 GPIO port.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;
;
; Inspired by the tinyForth by Luke Chang in Taiwan FIG Chapter
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;
```

4.1 FORTH Virtual Machine on MSP430G2553

msp430g2553.h contains all the register names and names of bits in these registers. It is included here first so that we can refer to the registers and bits with mnemonic names.

In the original eForth Model, a small group of FORTH commands were identified as kernel commands, low level commands, or primitive commands. These commands were coded in machine instructions of the host microprocessor. They allow the underlying microcontroller to become a FORTH Virtual Machine. All other commands were written as lists of commands, and are called high level commands or compound commands. Compound commands are lists of primitive commands and other compound commands. This division of commands was very useful in porting eForth to many different microprocessors, because only primitive commands needed to be rewritten when moving eForth to a new microprocessor.

In 430eForth, we retained this division. However, we use the Subroutine Threading Model and optimize many compound commands so that the system executes at the highest speed and occupies the least memory space. All commands that can be optimized are re-coded in assembly.

The CPU registers are assigned various functions required in a FORTH Virtual Machine (FVM) as follows:

Register	FVM Name	Function
R0(PC)		Program counter
R1(SP)		Return stack pointer
R2(SR)		Status register
R3		Constant generator
R4	tos	Top of parameter stack
R5	stack	Parameter stack pointer
R6	temp0	Scratch pad
R7	temp1	Scratch pad
R8	temp2	Scratch pad
R9	temp3	Scratch pad
R10		Not used
R11		Not used
R12		Not used
R13		Not used
R14	r14	UART delay counter
R15	r15	UART delay

```
;; CPU registers
tos .equ   R4
stack  .equ   R5
temp0  .equ   R6
temp1  .equ   R7
temp2  .equ   R8
temp3  .equ   R9
;; R14-15 used by software UART
```

Assembly Macros

LOADTOS	Pop the external parameter stack and copy the popped item into tos register. It is used to implement DROP commands, and many other commands consuming the top two items on the parameter stack. It uses stack register in post-increment addressing mode
SAVETOS	Push the top item on the parameter stack, which is cached in tos register, on the external parameter stack. It is used to implement DUP command, and commands which pushes new data on the parameter stack. It uses stack register in the pre-decrement addressing mode.

```
loadtos   .macro
   mov.w  @stack+,tos
   .endm

savetos   .macro
   decd.w stack
   mov.w  tos,0(stack)
   .endm;; Constants
```

Constants Used by Assembler

Constant	Value	Function
COMPO	$40	Lexicon compile-only bit
IMEDD	$80	Lexicon immediate bit
CELLL	2	Size of a cell in bytes
BASEE	10	Default radix for number conversion
BKSPP	8	Back space ASCII character
LF	10	Line feed ASCII character
CRR	13	Carriage return ASCII character
CALLL	$12B0	Machine code of call instruction
UPP	$200	Start of user area
DPP	$220	Start of free RAM space
SPP	$378	Top of parameter stack (SP0)
TIBB	$380	Terminal input buffer (TIB)
RPP	$3F8	Top of return stack (RP0)
CODEE	$C000	Start of FORTH dictionary
COLDD	$FFFE	Reset vector
EM	$FFFF	Top of flash main memory

Flash memory allocation of 430eForth in bytes:

Address	Contents
$1000	Information flash memory, Segment D
$C000	Start of FORTH dictionary
$FFFE	End of FORTH dictionary
$FFFF	End of flash memory

RAM memory allocation of 430eForth in bytes:

Address	Contents
$0	Special function and I/O registers
$200	System variables
$220	Free RAM space
$270	Initial PAD for number conversions
$378	Top of parameter stack
$380	Terminal input buffer
$3F8	Top of return stack

```
COMPO  .equ   040H   ;lexicon compile only bit
IMEDD  .equ   080H   ;lexicon immediate bit
MASKK  .equ   07F1FH ;lexicon bit mask
CELLL  .equ   2      ;size of a cell
BASEE  .equ   10     ;default radix
VOCSS  .equ   8      ;depth of vocabulary stack
BKSPP  .equ   8      ;backspace
LF     .equ   10     ;line feed
CRR    .equ   13     ;carriage return
ERR    .equ   27     ;error escape
TIC    .equ   39     ;tick
CALLL  .equ   012B0H ;NOP CALL opcodes

UPP    .equ   200H
DPP    .equ    220H
SPP    .equ   378H      ;data stack
TIBB   .equ   380H      ;terminal input buffer
RPP    .equ   3F8H      ;return stacl
CODEE  .equ   0C000H    ;code dictionary
COLDD  .equ   0FFFEH    ;cold start vector
EM     .equ   0FFFFH    ;top of memory
```

4.2 Startup Code

Flash memory location 0FFFEH is allocated for a reset vector. The reset vector contains an address pointing to the reset routine main. When MSP430G2553 boots up, it jumps to main and starts running. It first initializes the return stack pointer sp, the parameter stack pointer stack, and the top of stack tos. It uses the default internal

clock DCOCLK at about 1.1 MHz. The Sub Main Clock SMCLK is derived from DCOCLK, divided by 2, and will be used by the flash memory controller to read and write the flash memory.

It then executes the command IO!, and falls into a waiting loop, waiting the user to type a “B” character on the keyboard. When it receives a “B” character, it determines the UART baud rate for the software UART. It the jumps to the eForth cold boot routine COLD, which starts the eForth text interpreter to execute commands typed in by the user.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;
;; Main entry points and COLD start data
   .text
main:
init:
   nop                        ; main program
   mov    #RPP,SP                ; set up stack
   clr    tos
   mov    #SPP,stack
   mov.w  #WDTPW+WDTHOLD,&WDTCTL  ; Stop watchdog timer
   mov    #FWKEY+FSSEL1+FN0,&FCTL2 ; SMCLK/2
   bis.b  #043h,&P1DIR          ; P1.0 output
;  call  #DIAGNOSE
   call   #STOIO
;setup2
;  call   #KEY
;  call   #EMIT
;  jmp setup2
   br     #COLD
```

4.3 Device Dependent I/O

MSP430G2553 on the LaunchPad does not have an external clock. It runs on the DCOCLK. This clock is not accurate enough to generate baud rate clocks for a UART. 430eForth therefore includes a software UART which can lock to an external UART device by detecting a “B” character from the external UART. When 430eForth boots up, if falls into a waiting loop for the “B” character. With the “B” character, it calculates the baud rate of the external UART and uses this baud rate to

transmit and receive characters.

The software UART uses P1.1 pin to transmit and P1.2 pin to receive.

KEY	Wait until a character is received from the RX line of UART. The ASCII code of the received character is returned on stack.
EMIT	Transmit a character to TX line of UART.
!IO	Initialize software UART. Wait for "B" character received from RX line. Determine the baud rate of UART.
Delay	Delay 1 bit time for UART transmitter and receiver. A loop count is stored in R15 register, as determined by !IO. This count is copied from R15 to R14, and R14 is decremented to 0. At 2400 baud, the loop count is 61 when the master clock DCOCLK is running at 1.1 MHz.

```
;; Device dependent I/O

;   KEY( -- c )
;   Return input character.
    .word  0
    .byte  3,"KEY"
KEY
    savetos
    clr tos       ;receiver buffer
key1
    bit.b  #4,&P1IN     ;wait for start bit
    jnz key1
;   bis.b  #1,&P1OUT    ;turn on red LED
    mov r15,r14
    rra r14
    call   #delay1          ;delay half bit time
    mov #8,temp0
key2   call   #delay        ;
    bit.b  #4,&P1IN
    rrc.b   tos
key3   dec temp0
    jnz key2
    call   #delay       ;stop bit
;   bic.b  #1,&P1OUT    ;turn off red LED
    ret
```

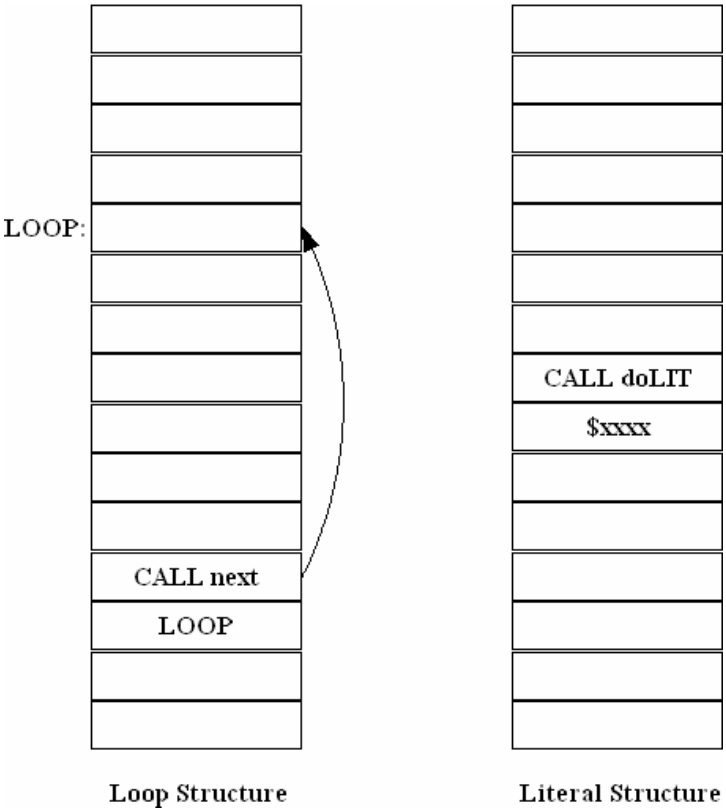
```
delay  mov r15,r14
delay1
    bit.b  #4,&P1IN
    dec r14
    jnz delay1
    ret

;   EMIT   ( c -- )
;   Send character c to the output device.
    .word  KEY-4
    .byte  4,"EMIT",0
EMIT
;   bis.b  #40h,&P1OUT      ;turn on green LED
    bic.b  #2,&P1OUT
    mov #8,temp0      ;send 8 data bits
emit1  call   #delay        ;start bit
    rrc.b  tos        ;shift LSB to carry
    jc  emit2
    bic.b  #2,&P1OUT
    jmp emit3
emit2
    bis.b  #2,&P1OUT
emit3  dec temp0
    jnz emit1
    call   #delay        ;last bit
    bis.b  #2,&P1OUT     ;idle TXD
    call   #delay        ;stop bit
;   bic.b  #40h,&P1OUT      ;turn off green LED
    loadtos
    ret

;   !IO( -- )
;   Initialize the serial I/O devices.
    .word  EMIT-6
    .byte  3,"!IO"
STOIO
    clr r15       ;wait for a "B" character from receiver
    bis.b  #043h,&P1OUT        ;idle, TXD, turn on both LED"s
iosto1
```

```
    bit.b  #4,&P1IN      ;wait for start bit
    jnz iosto1
    bic.b  #041h,&P1OUT        ;idle TXD, turn off both LED"s
iosto2 inc r15
    bit.b  #4,&P1IN             ;wait for a "B"
    jz  iosto2               ;R15 has count for 2 bittime
    rra r15       ;1 bittime
    bic.b  #041H,&P1OUT     ;turn off LED"s
    ret
```

4.4 Kernel

doLIT	Start a literal structures in compound commands. It allows numbers to be pushed on the parameter stack when the compound command is executed.
next	Terminate an indexed loop structures in compound command. A loop starts when the loop index is pushed on the return stack. When next is executed, it decrements this loop index on the return stack. If resulting index is not negative, jump back to repeat the loop. If the resulting index is negative, pop the return stack to discard the index, and exit the loop.

The literal structure and the indexed loop structure are show in the following figure:

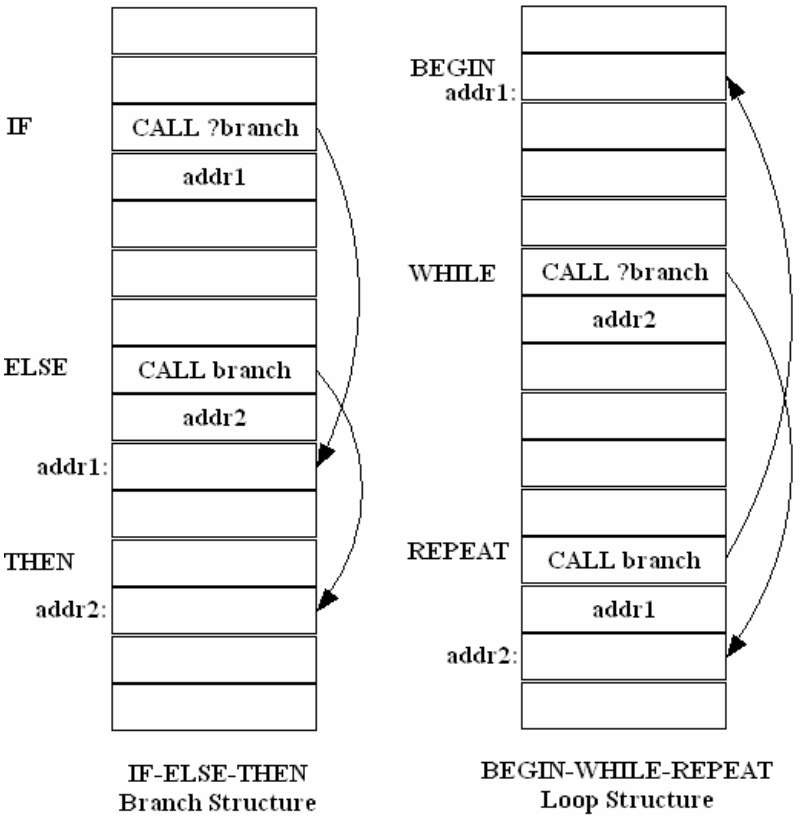



```
;; The kernel

;   doLIT  ( -- w )
;   Push an inline literal.
    .word  STOIO-4
    .byte  COMPO+5,"doLIT"
DOLIT
    savetos
    pop temp0
    mov @temp0+,tos
    br  temp0

;   EXIT   ( -- )
;   Terminate a colon definition.
    .word  DOLIT-6
    .byte  4,"EXIT"
EXIT
    pop temp0
    ret
```

```
;   EXECUTE   ( ca -- )
;   Execute the word at ca.
    .word  EXIT-6
    .byte  7,"EXECUTE"
EXECU
    mov tos,temp0
    loadtos
    br  temp0

;   next   ( -- )
;   Run time code for the single index loop.
;   : next ( -- ) \ hilevel model
;     r> r> dup if 1 - >r @ >r exit then drop cell+ >r ;
    .word  EXECU-8
    .byte  COMPO+4,"next",0
DONXT
    pop temp0
       dec    0(SP)   ;decrement index
    jge NEXT1
    pop temp1  ;discard index
    incd   temp0
    br  temp0
NEXT1:
    br  @temp0
```

Flow Control

?branch and branch commands are used to build control structures and loop structures in compound commands. In the following figure, an IF-ELSE-THEN branch structure and a BEGIN-WHILE-REPEAT loop structure are illustrated:



?branch	Build a conditional branch in compound commands.
branch	Build an unconditional branch in compound commands.
EXECUTE	Jump to an execution address on the top of the parameter stack. As the execution address is a byte address, it must be converted to a cell address for jumping. The cell address is pushed on the return stack and a RET instruction is executed to cause the jump.
EXIT	Terminate a compound command. Since it is executed as a call EXIT command, the return address must be popped off the return stack and then a ret instruction is executed. It is retained for compatibility. The call EXIT command can be simply replaced by a ret machine instruction.

```
;   ?branch   ( f -- )

;  Branch if flag is zero.
    .word  DONXT-6
    .byte  COMPO+7,"?branch"
QBRAN
    pop temp0
    bit #0xFFFF,tos
```

```
    loadtos
    jz  BRAN1
    incd   temp0
    br  temp0

;   branch ( -- )
;   Branch to an inline address.
    .word  QBRAN-8
    .byte  COMPO+6,"branch",0
BRAN
    pop temp0
BRAN1:
    br  @temp0
```

RAM Memory Access

MSP430G2553 has separated RAM memory and flash memory. The same set of memory read commands can be used to read either RAM or flash memory. However, a different set of commands is necessary to write to flash memory. The flash memory writing commands will be discussed later in the compiler section.

@	Read a 16-bit data stored in the address on top of the parameter stack. The address is a byte address pointing to a location in RAM memory.
!	Store the 16-bit data as the second item on parameter stack into the address on top of the parameter stack.
C@	Read an 8-bit data stored in the address on top of the parameter stack.
C!	Store an 8-bit data as the second item on parameter stack into the address on top of the parameter stack.

These 4 memory commands access data stored in RAM memory. Since in MSP430G2553, the I/O registers are mapped to the RAM memory space from 0 to $1FF, we can control MSP430G2553 interactively using these commands. This is the greatest advantage 430eForth has over the C/C++ programming environment which is a Compile-Load-Test no-interactive system.

To write flash memory, we have the `I!, ERASE`, and `WRITE` commands. They are discussed in a later section.

```
;   !  ( w a -- )
;   Pop the data stack to memory.
    .word  BRAN-8
    .byte  1,"!"
STORE
```

```
    mov.w  @stack+,0(tos)
    mov.w  @stack+,tos
    ret

;   @  ( a -- w )
;   Push memory location to the data stack.
    .word  STORE-2
    .byte  1,"@"
AT
    mov.w  @tos,tos
    ret

;   C! ( c b -- )
;   Pop the data stack to byte memory.
    .word  AT-2
    .byte  2,"C!",0
CSTOR
    mov.b  @stack+,0(tos)
    inc    stack
    mov.w  @stack+,tos
    ret

;   C@ ( b -- c )
;   Push byte memory location to the data stack.
    .word  CSTOR-4
    .byte  2,"C@",0
CAT
    mov.b  @tos,tos
    ret
```

Return Stack

430eForth system uses the return stack for two specific purposes: to save addresses while recursing through a token list, and to store the loop index for a FOR-NEXT loop.

Return stack is used by the FORTH Virtual Machine to save return addresses to be processed later. It is also a convenient place to store data temporarily. The return stack can thus be considered as an extension of the parameter stack. However, one must be very careful in using the return stack for temporary storage. The data pushed on the return stack must be popped off before ret is executed. Otherwise, ret

will get the wrong address to return to, and the system generally will crash. Since >R and R> are very dangerous to use, they are designed as compile-only commands and you can only use them in the compiling mode.

In setting up a loop, FOR compiles >R, which pushes the loop index from the parameter stack to the return stack. Inside the FOR-NEXT loop, the running index can be recalled by R@. NEXT compiles call next with an address after FOR. When next is executed, it decrements the loop index on the top of the return stack. If the index becomes negative, the loop is terminated; otherwise, next jumps back to the command after FOR.

>R	Pop a number off the parameter stack and pushes it on the return stack.
R>	Pop a number off the return stack and pushes it on the parameter stack.
R@	Copy the top item on the return stack and pushes it on the parameter stack without disturbing the return stack
SP@	Push the current parameter stack pointer on top of parameter stack. It is used to determine the depth of parameter stack.

```
;   R> ( -- w )
;   Pop the return stack to the data stack.
    .word  CAT-4
    .byte  2,"R",3EH,0
RFROM
    savetos
    pop temp0
    pop tos
    br  temp0

;   R@ ( -- w )
;   Copy top of return stack to the data stack.
    .word  RFROM-4
    .byte  2,"R@",0
RAT
    savetos
    pop temp0
    pop tos
    push   tos
    br  temp0

;   >R ( w -- )
;   Push the data stack to the return stack.
    .word  RAT-4
```

```
    .byte  COMPO+2,">R",0
TOR
    pop temp0
    push   tos
    loadtos
    br  temp0

;   SP@( -- a )
;   Push the current data stack pointer.
    .word  TOR-4
    .byte  3,"SP@"
SPAT:
    mov.w   stack,temp0
       savetos
    mov.w  temp0,tos
    ret
```

Parameter Stack

The parameter stack is the central location where all numerical data are processed, and where parameters are passed from one command to another. The stack items have to be arranged properly so that they can be retrieved in the Last-In-First-Out (LIFO) manner. When stack items are out of order, they can be rearranged by the stack words `DUP, SWAP, OVER` and `DROP`. There are other stack words useful in manipulating stack items, but these four are considered to be the minimum set.

DROP	Pop the parameter stack discards the top item on it.
DUP	Duplicate the top item and pushes it on the parameter stack.
SWAP	Exchange the two two item on the parameter stack.
OVER	Duplicates the second item and pushes it on the parameter stack.

```
;   DROP   ( w -- )
;   Discard top stack item.
    .word  SPAT-4
    .byte  4,"DROP",0
DROP
    loadtos
    ret

;   DUP( w -- w w )
```

```
;   Duplicate the top stack item.
    .word  DROP-6
    .byte  3,"DUP"
DUPP
    savetos
    ret

;   SWAP   ( w1 w2 -- w2 w1 )
;   Exchange top two stack items.
    .word  DUPP-4
    .byte  4,"SWAP",0
SWAP
    mov.w  tos,temp0
    mov.w  @stack,tos
    mov.w  temp0,0(stack)
    ret

;   OVER   ( w1 w2 -- w1 w2 w1 )
;   Copy second stack item to top.
    .word  SWAP-6
    .byte  4,"OVER",0
OVER
    mov.w  @stack,temp0
    savetos
    mov.w  temp0,tos
    ret
```

Logic

The only primitive command which cares about logic is `?branch`. It tests the top item on the stack. If it is zero, `?branch` will branch to the following address. If it is not zero, `?branch` will ignore the address and execute the command after the branch address. Thus we distinguish two logic values, zero for false and non-zero for true. Numbers used this way are called logic flags which can be either true or false. Logic flags thus cause conditional branching in control structures.

0<	Examine the top item on the parameter stack for its negativeness. If it is negative, 0< will return a -1 for true. If it is 0 or positive, 0< will return a 0 for false.
AND	Remove top two items on the parameter stack and pushes their bitwise logic AND results on the parameter stack.

OR	Remove top two items on the parameter stack and pushes their bitwise logic OR results on the parameter stack.
XOR	Remove top two items on the parameter stack and pushes their bitwise logic exclusive OR results on the parameter stack.
UM+	Add top two unsigned number on the data stack and replaces them with the unsigned sum of these two numbers and a carry on top of the sum. FORTH does not have access to the carry flag in MSP430G2553 CPU, and UM+ preserves the carry flag to be used in double integer arithmetic operations. In 430eForth, most arithmetic commands are coded in assembly and UM+ is not used often.

```
;  0< ( n -- t )
;  Return true if n is negative.
   .word  SWAP-6
   .byte  2,"0",3CH,0
ZLESS
   tst    tos
   mov#0xFFFF,tos
   jn  ZLESS1
   clr    tos
ZLESS1:
   ret

;  AND( w w -- w )
;  Bitwise AND.
   .word  ZLESS-4
   .byte  3,"AND"
ANDD
   and    @stack+,tos
   ret

;  OR ( w w -- w )
;  Bitwise inclusive OR.
   .word  ANDD-4
   .byte  2,"OR",0
ORR
   bis    @stack+,tos
   ret

;  XOR( w w -- w )
;  Bitwise exclusive OR.
```

```
    .word  ORR-4
    .byte  3,"XOR"
XORR
    xor    @stack+,tos
    ret

;   UM+( w w -- w cy )
;   Add two numbers, return the sum and carry flag.
    .word  XORR-4
    .byte  3,"UM+"
UPLUS
    clr    temp0
    add    @stack,tos
    rlc    temp0
    mov    tos,0(stack)
    mov    temp0,tos
    ret
```

4.5 System Variables

In 430eForth, all variables used by the system are merged together and are called system variables. They are allocated in a RAM memory array starting from location $200. They are all initialized by copying a table of initial values stored in flash information memory, Segment D, starting from location $1000.

When you finish a application, copy these variables back to Segment D, and the application, hopefully, will boot up on reset.

Variable	Address	Function
'BOOT	200H	Execution vector to start application command.
BASE	202H	Radix base for numeric conversion.
tmp	204H	Scratch pad.
HLD	206H	Pointer to a buffer holding next digit for numeric conversion.
>IN	208H	Input buffer character pointer used by text interpreter.
#TIB	20AH	Number of characters in input buffer.
'TIB	20CH	Address of Terminal Input Buffer.
'EVAL	20EH	Execution vector switching between $INTERPRET and $COMPILE.
CONTEXT	210H	Vocabulary array pointing to last name fields of dictionary.
CP	212H	Pointer to top of dictionary, the first available flash memory location to compile new command

DP	214H	Pointer to the first available RAM memory location.
LAST	216H	Pointer to name field of last command in dictionary.

```
;; System and user variables

;   BASE   ( -- a )
;   Storage of the radix base for numeric I/O.
    .word  UPLUS-4
    .byte  4,"BASE",0
BASE
    savetos
    mov #202H,tos
    ret

;   tmp( -- a )
;   A temporary storage location used in parse and find.
    .word  BASE-6
    .byte  COMPO+3,"tmp"
TEMP
    savetos
    mov #204H,tos
    ret

;   #TIB   ( -- a )
;   Hold the character pointer while parsing input stream.
    .word  TEMP-4
    .byte  4,"#TIB",0
NTIB
    savetos
    mov #206H,tos
    ret

;   >IN( -- a )
;   Hold the character pointer while parsing input stream.
    .word  NTIB-6
    .byte  3,">IN"
INN
```

```
    savetos
    mov #208H,tos
    ret

;   HLD( -- a )
;   Hold a pointer in building a numeric output string.
    .word  INN-4
    .byte  3,"HLD"
HLD
    savetos
    mov #20AH,tos
    ret

;   'EVAL  ( -- a )
;   A area to specify vocabulary search order.
    .word  HLD-4
    .byte  7,"'EVAL"
TEVAL
    savetos
    mov #20CH,tos
    ret

;   CONTEXT   ( -- a )
;   A area to specify vocabulary search order.
    .word  TEVAL-6
    .byte  7,"CONTEXT"
CNTXT
    savetos
    mov #20EH,tos
    ret

;   CP ( -- a )
;   Point to the top of the code dictionary.
    .word  CNTXT-8
    .byte  2,"CP",0
CP
    savetos
    mov #210H,tos
```

```
    ret

;   DP ( -- a )
;   Point to the bottom of the free ram area.
    .word  CP-4
    .byte  2,"DP",0
DP
    savetos
    mov#212H,tos
    ret

;   LAST  ( -- a )
;   Point to the last name in the name dictionary.
    .word  DP-4
    .byte  4,"LAST",0
LAST
    savetos
    mov#214H,tos
    ret
```

4.6 Common Functions

Arithmetic

This group of FORTH commands are commonly used in writing FORTH applications.

?DUP	Duplicate the top item on the parameter stack if it is non-zero.
ROT	Rotate the top three items on the parameter stack. The third item is pulled out to the top. The second item is pushed down to the third item, and the top item is pushed down to be the second item. ROT is unique in that it accesses the third item on the parameter stack. All other stack commands can only access one or two stack items. In FORTH programming, it is generally accepted that one should not try to access stack items deeper than the third item. When you have to access deeper into the data stack, it is a good time to re-evaluate your algorithm. Most often, you can avoid this situation by factoring your code into smaller parts which do not reach so deep into the parameter stack.
2DROP	Discard the top two items on the parameter stack.
2DUP	Duplicate the top two items on the parameter stack.
+	Add the top item on the parameter to the second item, and then pops the top item off the parameter stack. It is recoded in assembly for speed.

INVERT	Invert each individual bit in the top item on the parameter stack. It is often called 1's complement operation.
NEGATE	Negate the top item on the parameter stack. It is often called 2's complement operation.
DNEGATE	Negate the top two items on the parameter stack, as a 32-bit double integer.
-	Subtract the top item on the parameter stack from the second item, and then pops the top item off the parameter stack.
ABS	Replace the top item on the parameter stack with its absolute value.

```
;; Common functions

;   ?DUP   ( w -- w w | 0 )
;   Dup tos if its is not zero.
    .word  LAST-6
    .byte  4,"?DUP",0
QDUP
    tst    tos
    jnz    DUPP
    ret

;   ROT( w1 w2 w3 -- w2 w3 w1 )
;   Rot 3rd item to top.
    .word  QDUP-6
    .byte  3,"ROT"
ROT
    call   #TOR
    call   #SWAP
    call   #RFROM
    call   #SWAP
    ret

;   2DROP  ( w w -- )
;   Discard two items on stack.
    .word  ROT-4
    .byte  5,"2DROP"
DDROP
    call   #DROP
    CALL   #DROP
    ret
```

```
;   2DUP   ( w1 w2 -- w1 w2 w1 w2 )
;   Duplicate top two items.
    .word  DDROP-6
    .byte  4,"2DUP",0
DDUP
    call   #OVER
    call   #OVER
    ret

;   +  ( w w -- sum )
;   Add top two items.
    .word  DDUP-6
    .byte  1,"+"
PLUS
    add    @stack+,tos
    ret

;   D+ ( d d -- d )
;   Double addition, as an example using UM+.
;
    .word  PLUS-2
    .byte  2,"D+",0
DPLUS
;   call   #TOR
    call   #SWAP
    call   #TOR
    call   #UPLUS
;   call   #RFROM
    call   #RFROM
    call   #PLUS
    call   #PLUS
    ret

;   NOT( w -- w )
;   One's complement of tos.
    .word  DPLUS-4
    .byte  3,"NOT"
```

```
INVER
   inv tos
   ret

;  2/ ( w -- w )
;  Divide by 2.
   .word  INVER-4
   .byte  2,"2/",0
TWOSL
   rra tos
   ret

;  NEGATE ( n -- -n )
;  Two's complement of tos.
   .word  INVER-4
   .byte  6,"NEGATE",0
NEGAT
   inv tos
   inc tos
   ret

;  DNEGATE   ( d -- -d )
;  Two's complement of top double.
   .word  NEGAT-8
   .byte  7,"DNEGATE"
DNEGA
   call   #INVER
   call   #TOR
   call   #INVER

   call   #DOLIT
   .word  1
   call   #UPLUS
   call   #RFROM
   call   #PLUS
   ret

;  -  ( n1 n2 -- n1-n2 )
```

```
;  Subtraction.
   .word  DNEGA-8
   .byte  1,"-"
SUBB
   sub    @stack+,tos
   inv tos
   inc    tos
   ret

;  ABS( n -- n )
;  Return the absolute value of n.
   .word  SUBB-2
   .byte  3,"ABS"
ABSS
   call   #DUPP
   call   #ZLESS
   call   #QBRAN
   .word  ABS1
   call   #NEGAT
ABS1:  ret
```

Comparison

The primitive comparison commands in 430eForth are ?branch and 0<. However, ?branch is at such a low level that it is not used in compound commands. ?branch is secretly compiled into compound commands by IF as an address literal. For all intentions and purposes, we can consider IF the equivalent of ?branch. When IF is encountered, the top item on the parameter stack is considered a logic flag. If it is true (non-zero), the execution continues until ELSE, then jump to THEN, or to THEN directly if there is no ELSE clause.

The following logic words are constructed using the IF...ELSE...THEN structure with 0< and XOR. XOR is used as a "not equal" operator, because if the top two items on the parameter stack are not equal, the XOR operator will return a non-zero number, which is considered to be true.

=	Compare top two items on the parameter stack. If they are equal, replace these two items with a true flag; otherwise, replace them with a false flag.
U<	Compare two unsigned numbers on the top of the parameter stack. If the top item is less than the second item in unsigned comparison, replace these two items with a true flag; otherwise, replace them with a false flag. This command is very important, especially in comparing addresses, as we assume that the addresses are unsigned numbers pointing to unique memory locations. The arithmetic comparison operator < cannot be used

	to determine whether one address is higher or lower than the other. Using < for address comparison had been the single cause of many failures in the annals of FORTH. We don not have this problem in MSP430G2553 since it has only 32 KB of flash memory. However, watch out when you move 430eForth to a bigger chip.
<	Compare two signed numbers on the top of the parameter stack. If the top item is less than the second item in signed comparison, replace these two items with a true flag; otherwise, replace them with a false flag.
MAX	Retain the larger of the top two items on the parameter stack. Both numbers are assumed to be signed integers.
MIN	Retain the smaller of the top two items on the parameter stack. Both numbers are assumed to be signed integers.
WITHIN	Check whether the third item on the parameter stack is within the range as specified by the top two numbers on the parameter stack. The range is inclusive as to the lower limit and exclusive to the upper limit. If the third item is within range, a true flag is returned on the parameter stack, replacing all three items. Otherwise, a false flag is returned. All numbers are assumed to be signed integers.

```
;   =  ( w w -- t )
;  Return true if top two are equal.
    .word  ABSS-4
    .byte  1,3DH
EQUAL
    call   #XORR
    call   #QBRAN
    .word  EQU1
    call   #DOLIT
    .word  0
    ret ;false flag
EQU1:  call   #DOLIT
    .word  -1
    ret ;true flag

;   U< ( u u -- t )
;  Unsigned compare of top two items.
    .word  EQUAL-2
    .byte  2,"U",3CH,0
ULESS
    mov    @stack+,temp0
    cmp tos,temp0
    subc    tos,tos
```

```
    ret

;   <  ( n1 n2 -- t )
;   Signed compare of top two items.
    .word  ULESS-4
    .byte  1,3CH
LESS
    call   #DDUP
    call   #XORR
    call   #ZLESS
    call   #QBRAN
    .word  LESS1
    call   #DROP
    call   #ZLESS
    ret
LESS1: call   #SUBB
    call   #ZLESS
    ret

;   MAX( n n -- n )
;   Return the greater of two top stack items.
    .word  LESS-2
    .byte  3,"MAX"
MAX
    call   #DDUP
    call   #LESS
    call   #QBRAN
    .word  MAX1
    call   #SWAP
MAX1:  call   #DROP
    ret

;   MIN( n n -- n )
;   Return the smaller of top two stack items.
    .word  MAX-4
    .byte  3,"MIN"
MIN
    call   #DDUP
```

```
    call   #SWAP
    call   #LESS
    call   #QBRAN
    .word  MIN1
    call   #SWAP
MIN1:  call   #DROP
    ret

;   WITHIN ( u ul uh -- t )
;   Return true if u is within the range of ul and uh.
    .word  MIN-4
    .byte  6,"WITHIN",0
WITHI
    call   #OVER
    call   #SUBB
    call   #TOR      ;ul <= u < uh
    call   #SUBB
    call   #RFROM
    call   #ULESS
    ret
```

Divide

UM/MOD and UM* are the most complicated and comprehensive division and multiplication commands. Once they are coded, all other division and multiplication operators can be derived easily from them. It has been a tradition in FORTH programming that one solves the most difficult problem first, and all other problems are solved by themselves.

UM/MOD	Divide an unsigned double integer by an unsigned single integer. It returns the unsigned remainder and unsigned quotient on the parameter stack. It is coded in assembly and the double integer dividend is stored in 4 registers temp0 to temp3. Division is carried out similar to long hand division.
M/MOD	Divide a signed double integer by a signed single integer. It returns the signed remainder and signed quotient on the parameter stack. The signed division is floored towards negative infinity.
/MOD	Divide a signed single integer by a signed integer. It replaces these two items with the signed remainder and quotient.
MOD	Divide a signed single integer by a signed integer. It replaces these two items with the signed remainder only.
/	Divide a signed single integer by a signed integer. It replaces these two

	items with the signed quotient only.

```
;; Divide

;   UM/MOD ( udl udh u -- ur uq )
;  Unsigned divide of a double by a single. Return mod and
quotient.
    .word  WITHI-8
    .byte  6,"UM/MOD",0
UMMOD
    call   #DDUP
    call   #ULESS
    call   #QBRAN
    .word  UMM4
    call   #NEGAT
    call   #DOLIT
    .word  15
    call   #TOR
UMM1:
    call   #TOR
    call   #DUPP
    call   #UPLUS
    call   #TOR
    call   #TOR
    call   #DUPP
    call   #UPLUS
    call   #RFROM
    call   #PLUS
    call   #DUPP
    call   #RFROM
    call   #RAT
    call   #SWAP
    call   #TOR
    call   #UPLUS
    call   #RFROM
    call   #ORR
    call   #QBRAN
    .word  UMM2
```

```
    call   #TOR
    call   #DROP
    add    #1,tos
    call   #RFROM
    call   #BRAN
    .word  UMM3
UMM2:
    call   #DROP
UMM3:
    call   #RFROM
    call   #DONXT
    .word  UMM1
    call   #DROP
    call   #SWAP
    ret
UMM4:
    call   #DROP
    call   #DDROP
    call   #DOLIT
    .word  -1
    call   #DUPP
    ret ;overflow, return max

;   M/MOD  ( d n -- r q )
;   Signed floored divide of double by single. Return mod and
quotient.
    .word  UMMOD-8
    .byte  5,"M/MOD"
MSMOD
    call   #DUPP
    call   #ZLESS
    call   #DUPP
    call   #TOR
    call   #QBRAN
    .word  MMOD1
    call   #NEGAT
    call   #TOR
    call   #DNEGA
```

```
    call   #RFROM
MMOD1:
    call   #TOR
    call   #DUPP
    call   #ZLESS
    call   #QBRAN
    .word  MMOD2
    call   #RAT
    call   #PLUS
MMOD2:
    call   #RFROM
    call   #UMMOD
    call   #RFROM
    call   #QBRAN
    .word  MMOD3
    call   #SWAP
    call   #NEGAT
    call   #SWAP
MMOD3: ret

;   /MOD   ( n n -- r q )
;   Signed divide. Return mod and quotient.
    .word  MSMOD-6
    .byte  4,"/MOD",0
SLMOD
    call   #OVER
    call   #ZLESS
    call   #SWAP
    call   #MSMOD
    ret

;   MOD( n n -- r )
;   Signed divide. Return mod only.
    .word  SLMOD-6
    .byte  3,"MOD"
MODD
    call   #SLMOD
    call   #DROP
```

```
    ret

;   /  ( n n -- q )
;   Signed divide. Return quotient only.
    .word  MODD-4
    .byte  1,"/"
SLASH
    call   #SLMOD
    call   #SWAP
    call   #DROP
    ret
```

Multiply

UM*	Multiply two unsigned single integers and returns the unsigned double integer product on the parameter stack. `UM*` command takes advantage of the multiply machine instructions in MSP430G2553 chip. The multiply instructions in MSP430G2553 operate on 8 bit values, and the 16 bit products have to be added properly to form a 32 bit double integer product.
*	Multiply two signed single integers and returns the signed single integer product on the parameter stack.
M*	Multiply two signed single integers and returns the signed double integer product on the parameter stack.
*/MOD	Multiply the signed integers `n1` and `n2`, and then divides the double integer product by `n3`. It in fact is ratioing `n1` by `n2/n3`. It returns both the remainder and the quotient.
*/	Multiply the signed integers `n1` and `n2`, and then divides the double integer product by `n3`. It returns only the quotient.

FORTH is very close to assembly languages in that it generally only handles integer numbers. There are floating point extensions in many more sophisticated FORTH systems, but they are more exceptions than rules. The reason why FORTH has traditionally been an integer language is that integers are handled faster and more efficiently in the computers, and most technical problems can be solved satisfactorily only using integers. A 16-bit integer has the dynamic range of 110 dB which is far more than enough for most engineering problems. The precision of a 16-bit integer representation is limited to one part in 65535, which could be inadequate for small numbers. However, the precision can be greatly improved by scaling; i.e., taking the ratio of two integers. It was demonstrated that pi, or any other irrational numbers, can be represented accurately to 1 part in 100,000,000 by a ratio of two 16-bit integers.

The scaling commands `*/MOD` and `*/` are useful in scaling number `n1` by the ratio of `n2/n3`. When `n2` and `n3` are properly chosen, the scaling commands can

preserve precision similar to the floating point operations at a much higher speed. Notice also that in these scaling operations, the intermediate product of `n1` and `n2` is a double precision integer so that the precision of scaling is maintained.

```
;; Multiply

;  UM*( u u -- ud )
;  Unsigned multiply. Return double product.
   .word  SLASH-2
   .byte  3,"UM*"
UMSTA
   call   #DOLIT
   .word  0
   call   #SWAP
   call   #DOLIT
   .word  15
   call   #TOR
UMST1: call   #DUPP
   call   #UPLUS
   call   #TOR
   call   #TOR
   call   #DUPP
   call   #UPLUS
   call   #RFROM
   call   #PLUS
   call   #RFROM
   call   #QBRAN
   .word  UMST2
   call   #TOR
   call   #OVER
   call   #UPLUS
   call   #RFROM
   call   #PLUS
UMST2: call   #DONXT
   .word  UMST1
   call   #ROT
   jmp    DROP

;  *  ( n n -- n )
```

```
;  Signed multiply. Return single product.
   .word  UMSTA-4
   .byte  1,"*"
STAR
   call   #UMSTA
   jmp    DROP

;  M* ( n n -- d )
;  Signed multiply. Return double product.
   .word  STAR-2
   .byte  2,"M*"
MSTAR
   call   #DDUP
   call   #XORR
   call   #ZLESS
   call   #TOR
   call   #ABSS
   call   #SWAP
   call   #ABSS
   call   #UMSTA
   call   #RFROM
   call   #QBRAN
   .word  MSTA1
   call   #DNEGA
MSTA1: ret

;  */MOD  ( n1 n2 n3 -- r q )
;  Multiply n1 and n2, then divide by n3. Return mod and
quotient.
   .word  MSTAR-4
   .byte  5,"*/MOD"
SSMOD
   call   #TOR
   call   #MSTAR
   call   #RFROM
   call   #MSMOD
   ret
```

```
;   */ ( n1 n2 n3 -- q )
;   Multiply n1 by n2, then divide by n3. Return quotient only.
    .word  SSMOD-6
    .byte  2,"*/"
STASL
    call   #SSMOD
    call   #SWAP
    call   #DROP
    ret
```

4.7 Miscellaneous

CELL+	Increment the top item on the parameter stack by 2.
CELL-	Decrement the top item on the parameter stack by 2.
ALIGNED	Modify the byte address on top of the parameter stack so that it points to the next word boundary.
BL	Push a blank or space character (ASCII 32) on parameter stack. `BL` is often used in parsing out space delimited strings.
>CHAR	Convert a non-printable character to a harmless underscore character(ASCII 95). As 430eForth is designed to communicate with a host computer through a serial I/O device, it is important that 430eForth will not emit control characters to the host and thereby causes unexpected behavior on the host computer. `>CHAR` thus filters the characters before they are sent out by `EMIT`.
DEPTH	Push the number of items currently on the parameter stack to the top of the stack.
PICK	Take a number `n` off the parameter stack and replaces it with the n'th item on the parameter stack. The number `n` is 0-based; i.e., the top item is number 0, the next item is number 1, etc. Therefore, `0 PICK` is equivalent to `DUP`, and `1 PICK` is equivalent to `OVER`.

```
;; Miscellaneous
;   CELL+ ( a -- a )
;   Add cell size in byte to address.
    .word  STASL-4
    .byte  5,"CELL+"
CELLP
    add    #2,tos
    ret

;   CELL- ( a -- a )
```

```
;  Subtract cell size in byte from address.
   .word  CELLP-6
   .byte  5,"CELL-"
CELLM
   sub    #2,tos
   ret

;  CELLS  ( n -- n )
;  Multiply tos by cell size in bytes.
   .word  CELLM-6
   .byte  5,"CELLS"
CELLS
   rla    tos
   ret

;  ALIGNED   ( b -- a )
;  Align address to the cell boundary.
   .word  CELLS-6
   .byte  7,"ALIGNED"
ALGND
   add#1,tos
   bic#1,tos
   ret

;  BL ( -- 32 )
;  Return 32, the blank character.
   .word  ALGND-8
   .byte  2,"BL",0
BLANK
   savetos
   mov#20H,tos
   ret

;  >CHAR  ( c -- c )
;  Filter non-printing characters.
   .word  BLANK-4
   .byte  5,">CHAR"
TCHAR
```

```
    call   #DUPP  ;mask msb
    call   #BLANK
    call   #DOLIT
    .word  127
    call   #WITHI ;check for printable
    call   #QBRAN
    .word  TCHA1
    ret
TCHA1:
    call   #DROP
    call   #DOLIT
    .word  "_";replace non-printables
    ret

;   DEPTH  ( -- n )
;   Return the depth of the data stack.
    .word  TCHAR-6
    .byte  5,"DEPTH"
DEPTH
    call   #SPAT
    call   #DOLIT
    .word  SPP
    call   #SWAP
    call   #SUBB
    jmp    TWOSL

;   PICK   ( ... +n -- ... w )
;   Copy the nth stack item to tos.
    .word  DEPTH-6
    .byte  4,"PICK",0
PICK
;   add    #1,tos
    call   #CELLS
    call   #SPAT
    call   #PLUS
    call   #AT
    ret
```

Memory Access

A memory array is generally specified by its starting address and its length in bytes.

In a count string, the first byte is a count byte, specifying the number of bytes in the following string. String literals in compound commands and the name strings in the headers of command records are all represented by count strings.

Following commands are useful in accessing memory arrays and strings.

+!	Add the second item on the parameter stack to the cell addressed by the top item on the stack.
COUNT	Fetch one byte from RAM memory pointed to by the address on the top of the parameter stack. This address is incremented by 1, and the byte just read is pushed on the stack. `COUNT` is designed to get the count byte at the beginning of a counted string, and returns the address of the first byte in the string and the length of this string. However, it is often used in a loop to read consecutive bytes in a byte array.
HERE	Push the address of the first free location in the RAM memory. FORTH text interpreter stores here a string parsed out of the Terminal Input Buffer and then searches the dictionary for a command with this name.
PAD	Push on the parameter stack the address of the text buffer where numbers to be output are constructed and text strings are stored temporarily. It is 64 bytes above `HERE`.
TIB	Push the address of the Terminal Input Buffer on the parameter stack. Terminal Input Buffer stores a line of text from the serial I/O input device. FORTH text interpreter then processes or interprets this line of text.
@EXECUTE	Fetch a code field address of a command which is stored in the address on the top of the parameter stack, and jumps to it to execute this command. It is used extensively to execute vectored commands stored in RAM memory. The behavior of a vectored command can be changed dynamically at the run time.
CMOVE	Copy a byte array from one location to another in RAM memory. The top three item on the parameter stack are the source address, the destination address and the number of bytes to be copied.
UPPER	Convert the ASCII character on the top of the parameter stack to an upper case character. This command is used to convert input text string to an upper case string so that the text interpreter is now case insensitive.
FILL	Fill a memory array with the same byte. The top three items on the parameter stack are the address of the array, the length of the array in bytes, and the byte value to be filled into this array.

```
;; Memory access
```

```
;  +! ( n a -- )
;  Add n to the contents at address a.
   .word  PICK-6
   .byte  2,"+!",0
PSTOR
   call   #SWAP
   call   #OVER
   call   #AT
   call   #PLUS
   call   #SWAP
   call   #STORE
   ret

;  2! ( d a -- )
;  Store the double integer to address a.
   .word  PSTOR-4
   .byte  2,"2!",0
DSTOR
   call   #SWAP
   call   #OVER
   call   #STORE
   call   #CELLP
   call   #STORE
   ret

;  2@ ( a -- d )
;  Fetch double integer from address a.
   .word  DSTOR-4
   .byte  2,"2@",0
DAT
   call   #DUPP
   call   #CELLP
   call   #AT
   call   #SWAP
   call   #AT
   ret
```

```
;   COUNT  ( b -- b +n )
;   Return count byte of a string and add 1 to byte address.
    .word  DAT-4
    .byte  5,"COUNT"
COUNT
    mov.b  @tos+,temp0
    savetos
    mov    temp0,tos
    ret

;   HERE   ( -- a )
;   Return the top of the code dictionary.
    .word  COUNT-6
    .byte  4,"HERE"
HERE
    call   #DP
    call   #AT
    ret

;   PAD( -- a )
;   Return the address of a temporary buffer.
    .word  HERE-6
    .byte  3,"PAD"
PAD
    call   #HERE
    add    #50,tos
    ret

;   TIB( -- a )
;   Return the address of the terminal input buffer.
    .word  PAD-4
    .byte  3,"TIB"
TIB
    Savetos
    Mov#TIBB,tos
    Ret

;   @EXECUTE  ( a -- )
```

```
;  Execute vector stored in address a.
   .word  TIB-4
   .byte  8,"@EXECUTE",0
ATEXE
   call   #AT
   call   #QDUP     ;?address or zero
   call   #QBRAN
   .word  EXE1
   call   #EXECU    ;execute if non-zero
EXE1:  ret    ;do nothing if zero

;  CMOVE  ( b1 b2 u -- )
;  Copy u bytes from b1 to b2.
   .word  ATEXE-10
   .byte  5,"CMOVE"
CMOVE
   call   #TOR
   call   #BRAN
   .word  CMOV2
CMOV1: call   #TOR
   call   #COUNT
   call   #RAT
   call   #CSTOR
   call   #RFROM,
   add    #1,tos
CMOV2: call   #DONXT
   .word  CMOV1
   call   #DDROP
   ret

;  FILL   ( b u c -- )
;  Fill u bytes of character c to area beginning at b.
   .word  CMOVE-6
   .byte  4,"FILL",0
FILL
   call   #SWAP
   call   #TOR
   call   #SWAP
```

```
    call   #BRAN
    .word  FILL2
FILL1: call   #DDUP
    call   #CSTOR
    add    #1,tos
FILL2: call   #DONXT
    .word  FILL1
    call   #DDROP
    ret
```

4.8 Input Output

Numeric Output

FORTH is interesting in its special capabilities in handling numbers across a man-machine interface. It recognizes that machines and humans prefer very different representations of numbers. Machines prefer binary representation, but humans prefer decimal Arabic representation. However, depending on circumstances, a human may want numbers to be represented in other radices, like hexadecimal, octal, and sometimes binary.

FORTH solves this problem of internal (machine) versus external (human) number representations by insisting that all numbers are represented in binary form in CPU and memory. Only when numbers are imported or exported for human consumption are they converted to external ASCII representation. The radix of the external representation is stored in system variable BASE. You can select any reasonable radix in BASE, up to 72, limited by available printable characters in the ASCII character set.

The output number string is built below the PAD buffer in RAM memory. The least significant digit is extracted from the integer on the top of the parameter stack by dividing it by the current radix in BASE. The digit thus extracted is added to the output string backwards from PAD to the low memory. The conversion is terminated when the integer is divided to zero. The address and length of the number string are made available by #> for outputting.

An output number conversion is initiated by <# and terminated by #>. Between them, # converts one digit at a time, #S converts all the digits, while HOLD and SIGN inserts special characters into the string under construction. This set of commands is very versatile and can handle all different output formats.

DIGIT	Convert an integer digit to the corresponding ASCII character.
EXTRACT	Extract the least significant digit from a number n on the top of the parameter stack. n is divided by the radix in BASE and the extracted digit is converted to its ASCII character which is pushed on the

	parameter stack.
<#	Initiate the output number onversion process by storing PAD buffer address into system variable HLD, which points to the location next numeric digit will be stored.
HOLD	Append an ASCII character whose code is on the top of the parameter stack, to the numeric out put string at HLD. HLD is decremented to receive the next digit.
#	Extract one digit from integer on the top of the parameter stack, according to radix in BASE, and add it to output numeric string.
#S	Extract all digits to output string until the integer on the top of the parameter stack is 0.
SIGN	Insert a - sign into the numeric output string if the integer on the top of the parameter stack is negative.
#>	Terminate the numeric conversion and pushes the address and length of output numeric string on the parameter stack.
str	Convert a signed integer on the top of the parameter stack to a numeric output string.
HEX	Set numeric conversion radix to 16 for hexadecimal conversions.
DECIMAL	Set numeric conversion radix to 10 for decimal conversions.

```
;; Numeric output, single precision
;  DIGIT ( u -- c )
;  Convert digit u to a character.
   .word  FILL-6
   .byte  5,"DIGIT"
DIGIT
   call   #DOLIT
   .word  9
   call   #OVER
   call   #LESS
   call   #DOLIT
   .word  7
   call   #ANDD
   call   #PLUS
   add    #"0",tos
;  call   #DOLIT
;  .word  "0"
;  call   #PLUS
   ret
```

```
;   EXTRACT   ( n base -- n c )
;   Extract the least significant digit from n.
    .word  DIGIT-6
    .byte  7,"EXTRACT"
EXTRC

    call   #DOLIT
    .word  0
    call   #SWAP
    call   #UMMOD
    call   #SWAP
    call   #DIGIT
    ret

;   <# ( -- )
;   Initiate the numeric output process.
    .word  EXTRC-8
    .byte  2,"<#",0
BDIGS
    call   #PAD
    call   #HLD
    call   #STORE
    ret

;   HOLD   ( c -- )
;   Insert a character into the numeric output string.
    .word  BDIGS-4
    .byte  4,"HOLD",0
HOLD
    call   #HLD
    call   #AT,
    sub    #1,tos
    call   #DUPP
    call   #HLD
    call   #STORE
    call   #CSTOR
    ret
```

```
;   #   ( u -- u )
;   Extract one digit from u and append the digit to output
string.
    .word  HOLD-6
    .byte  1,"#"
DIG
    call   #BASE
    call   #AT
    call   #EXTRC
    call   #HOLD
    ret

;   #S ( u -- 0 )
;   Convert u until all digits are added to the output string.
    .word  DIG-2
    .byte  2,"#S",0
DIGS
DIGS1:
    call   #DIG
    call   #DUPP
    call   #QBRAN
    .word  DIGS2
    call   #BRAN
    .word  DIGS1
DIGS2: ret

;   SIGN   ( n -- )
;   Add a minus sign to the numeric output string.
    .word  DIGS-4
    .byte  4,"SIGN",0
SIGN
    call   #ZLESS
    call   #QBRAN
    .word  SIGN1
    call   #DOLIT
    .word  "-"
    call   #HOLD
```

```
SIGN1: ret

;   #> ( w -- b u )
;   Prepare the output string to be TYPE'd.
    .word  SIGN-6
    .byte  2,"#",3EH,0
EDIGS
    call   #DROP
    call   #HLD
    call   #AT
    call   #PAD
    call   #OVER
    call   #SUBB
    ret

;   str( n -- b u )
;   Convert a signed integer to a numeric string.
    .word  EDIGS-4
    .byte  3,"str"
STR
    call   #DUPP
    call   #TOR
    call   #ABSS
    call   #BDIGS
    call   #DIGS
    call   #RFROM
    call   #SIGN
    call   #EDIGS
    ret

;   HEX( -- )
;   Use radix 16 as base for numeric conversions.
    .word  STR-4
    .byte  3,"HEX"
HEX
    call   #DOLIT
    .word  16
    call   #BASE
```

```
    call   #STORE
    ret

;   DECIMAL   ( -- )
;   Use radix 10 as base for numeric conversions.
    .word  HEX-4
    .byte  7,"DECIMAL"
DECIM
    call   #DOLIT
    .word  10
    call   #BASE
    call   #STORE
    ret
```

Numeric Input

The 430eForth text interpreter must handle numbers input to the system. It parses commands out of the input stream and executes them in sequence. When the text interpreter encounters a string which is not the name of a command in the dictionary, it assumes that the string must be a number and attempts to convert the ASCII digit string to a number according to the current radix. When the text interpreter succeeds in converting the string to a number, the number is pushed on the parameter stack for future use, if the text interpreter is in the interpreting mode. If it is in the compiling mode, the text interpreter will compile the number to the dictionary as an integer literal so that when the command under construction is later executed, the integer value will be pushed on the parameter stack.

If the text interpreter fails to convert the string to a number, this is an error condition which will cause the text interpreter to abort, post an error message to you, and then wait for your next line of commands.

DIGIT?	Convert an ASCII numeric digit c on the top of the parameter stack to its numeric value u according to current radix b. If conversion is successful, push a true flag above u. If not successful, return c and a false flag.
NUMBER?	Convert a count string of ASCII numeric digits at location a to an integer. If first character is a $, convert in hexadecimal; otherwise, convert using radix in BASE. If first character is a -, negate converted integer. If an illegal character is encountered, the address of string and a false flag are pushed on the parameter stack. Successful conversion pushes integer value and a true flag on the parameter stack. NUMBER? is very complicated because it has to cover many formats in the input numeric string. It also has to detect the error condition when it encounters an illegal numeric digit. .

```
;; Numeric input, single precision

;   DIGIT? ( c base -- u t )
;   Convert a character to its numeric value. A flag indicates
success.
    .word  DECIM-8
    .byte  6,"DIGIT?",0
DIGTQ
    call   #TOR,
    sub    #"0",tos
;   call   #DOLIT
;   .word  "0"
;   call   #SUBB
    call   #DOLIT
    .word  9
    call   #OVER
    call   #LESS
    call   #QBRAN
    .word  DGTQ1
    sub    #7,tos
    call   #DUPP,
    call   #DOLIT
    .word  10
    call   #LESS
    call   #ORR
DGTQ1: call   #DUPP
    call   #RFROM
    call   #ULESS
    ret

;   NUMBER?   ( a -- n T | a F )
;   Convert a number string to integer. Push a flag on tos.
    .word  DIGTQ-8
    .byte  7,"NUMBER?"
NUMBQ
    call   #BASE
    call   #AT
    call   #TOR,
```

```
    call   #DOLIT
    .word  0
    call   #OVER
    call   #COUNT
    call   #OVER
    call   #CAT,
    call   #DOLIT
    .word  "$"
      Call#EQUAL
    call   #QBRAN
    .word  NUMQ1
    call   #HEX
    call   #SWAP
    add    #1,tos
    call   #SWAP
    sub    #1,tos
NUMQ1: call   #OVER
    call   #CAT,
    call   #DOLIT
    .word  "-"
      Call#EQUAL
    call   #TOR
    call   #SWAP
    call   #RAT
    call   #SUBB
    call   #SWAP
    call   #RAT
    call   #PLUS
    call   #QDUP
    call   #QBRAN
    .word  NUMQ6
    sub    #1,tos
    call   #TOR
NUMQ2: call   #DUPP
    call   #TOR
    call   #CAT
    call   #BASE
    call   #AT
```

```
    call   #DIGTQ
    call   #QBRAN
    .word  NUMQ4
    call   #SWAP
    call   #BASE
    call   #AT
    call   #STAR
    call   #PLUS
    call   #RFROM
    add    #1,tos
    call   #DONXT
    .word  NUMQ2
    call   #RAT
    call   #SWAP
    call   #DROP
    call   #QBRAN
    .word  NUMQ3
    call   #NEGAT
NUMQ3: call   #SWAP
    call   #BRAN
    .word  NUMQ5
NUMQ4: call   #RFROM
    call   #RFROM
    call   #DDROP
    call   #DDROP,
    call   #DOLIT
    .word  0
NUMQ5: call   #DUPP
NUMQ6: call   #RFROM
    call   #DDROP
    call   #RFROM
    call   #BASE
    call   #STORE
    ret
```

Basic I/O

430eForth system assumes that it communicates with its environment only through a

serial I/O interface. To support the serial I/O, only three words are needed:

SPACE	Output a blank (space) character, ASCII 32.
CHARS	Output n ASCII characters. The ASCII code is on the top of the parameter stack, and number n is the second item on the parameter stack
SPACES	Output n blank (space) characters.
TYPE	Output n characters from a string in RAM memory. The second item on the parameter stack is the address of the string array, and the length in bytes is on the top of the parameter stack.
CR	Output a carriage-return and a line-feed, ASCII 13 and 10.

```
;; Basic I/O

;   SPACE  ( -- )
;   Send the blank character to the output device.
    .word  NUMBQ-8
    .byte  5,"SPACE"
SPACE
    call   #BLANK
    call   #EMIT
    ret

;   SPACES ( +n -- )
;   Send n spaces to the output device.
    .word  SPACE-6
    .byte  6,"SPACES",0
SPACS
    call   #DOLIT
    .word  0
    call   #MAX
    call   #TOR
    call   #BRAN
    .word  CHAR2
CHAR1: call   #SPACE
CHAR2: call   #DONXT
    .word  CHAR1
    ret

;   TYPE   ( b u -- )
;   Output u characters from b.
    .word  SPACS-8
```

```
    .byte  4,"TYPE",0
TYPEE
    call   #TOR
    call   #BRAN
    .word  TYPE2
TYPE1: call   #DUPP
    call   #CAT
    call   #TCHAR
    call   #EMIT
    add    #1,tos
TYPE2:
    call   #DONXT
    .word  TYPE1
    call   #DROP
    ret

;   CR ( -- )
;   Output a carriage return and a line feed.
    .word  TYPEE-6
    .byte  2,"CR",0
CR
    call   #DOLIT
    .word  CRR
    call   #EMIT
    call   #DOLIT
    .word  LF
    call   #EMIT
    ret
```

String literals are data structures compiled in compound command, in-line with other tokens, literal structures, and control structures. A string literal must start with a string token which knows how to handle the following string at run time. Here are two examples of string literals:

```
: xxx   ...   $" A compiled string"  ...   ;

: yyy   ...   ." An output string"  ...   ;
```

In compound command xxx, `$"` is an immediate command which compiles the following string as a string literal preceded by a special token `$"|`. When `$"|` is executed at run time, it returns the address of this string on the parameter stack. In

yyy, ." compiles a string literal preceded by another token `."|`, which prints the compiled string to the output device at run time.

do$	Push the address of a string literal on the parameter stack. It is called by a string token like $"| or ."|, which precede their respective strings in flash memory. Therefore, the second item on the return stack points to the string. This address is pushed on the parameter stack. This second item on the return stack must be modified so that it will point to the next token after the string literal. This way. the token after the string literal will be executed, skipping over the string literal. Both $"\| and ."\| use the word do$, which retrieve the address of a string stored as the second item on the return stack.
$"|	Push the address of the following string on the parameter stack, and then executes the token immediately following the string.
."|	Print the following string, and then executes the token immediately following the string.

```
;  do$( -- a )
;  Return the address of a compiled string.
   .word  CR-4
   .byte  COMPO+3,"do$"
DOSTR
   call   #RFROM
   call   #RAT
   call   #RFROM
   call   #COUNT
   call   #PLUS
   call   #ALGND
   call   #TOR
   call   #SWAP
   call   #TOR
   ret

;  $"|( -- a )
;  Run time routine compiled by $". Return address of a
compiled string.
   .word  DOSTR-4
   .byte  COMPO+3,"$""|"
STRQP
   call   #DOSTR
   ret ;force a call to do$
```

```
;   ."|( -- )
;   Run time routine of ." . Output a compiled string.
    .word  STRQP-4
    .byte  COMPO+3,""|"
DOTQP
    call   #DOSTR
    call   #COUNT
    call   #TYPEE
    ret
```

With the number formatting command set as shown above, one can format numbers for output in any format desired. The free output format is a number string preceded by a single space. The fix column format displays a number right-justified in a column of a pre-determined width. The commands `' .' , 'U.',` and `?` use the free format. The words `.R` and `U.R` use the fix format.

.R	Print a signed integer n , the second item on the parameter stack, right-justified in a field of +n characters. +n is on the top of the parameter stack.
U.R	Print an unsigned integer n right-justified in a field of +n characters.
U.	Print an unsigned integer u in free format, followed by a space.
.	Print a signed integer n in free format, followed by a space.
?	Print signed integer stored in memory a on the top of the parameter stack, in free format followed by a space.

```
;   .R ( n +n -- )
;   Display an integer in a field of n columns, right justified.
    .word  DOTQP-4
    .byte  2,".R",0
DOTR
    call   #TOR
    call   #STR
    call   #RFROM
    call   #OVER
    call   #SUBB
    call   #SPACS
    call   #TYPEE
    ret
```

```
;   U.R( u +n -- )
;   Display an unsigned integer in n column, right justified.
    .word  DOTR-4
    .byte  3,"U.R"
UDOTR
    call   #TOR
    call   #BDIGS
    call   #DIGS
    call   #EDIGS
    call   #RFROM
    call   #OVER
    call   #SUBB
    call   #SPACS
    call   #TYPEE
    ret

;   U. ( u -- )
;   Display an unsigned integer in free format.
    .word  UDOTR-4
    .byte  2,"U.",0
UDOT
    call   #BDIGS
    call   #DIGS
    call   #EDIGS
    call   #SPACE
    call   #TYPEE
    ret

;   .  ( w -- )
;   Display an integer in free format, preceded by a space.
    .word  UDOT-4
    .byte  1,"."
DOT
    call   #BASE
    call   #AT
    call   #DOLIT
```

```
    .word  10
    call   #XORR  ;?decimal
    call   #QBRAN
    .word  DOT1
    jmp    UDOT
DOT1:
    call   #STR
    call   #SPACE
    jmp    TYPEE

;   ?  ( a -- )
;   Display the contents in a memory cell.
    .word  DOT-2
    .byte  1,"?"
QUEST
    call   #AT
    call   #DOT
    ret
```

4.9 Parsing

Parsing is always considered a very advanced topic in computer science. However, because FORTH uses very simple syntax rules, parsing is easy. FORTH input stream consists of ASCII strings separated by spaces and other white space characters like tabs, carriage returns, and line feeds. The text interpreter scans the input stream, parses out strings, and interprets them in sequence. After a string is parsed out of the input stream, the text interpreter will 'interpret' it; i.e., execute it if it is a valid command, compile it if the text interpreter is in the compiling mode, and convert it to a number if the string is not a FORTH command.

The case where the delimiting character is a space (ASCII 32) is special, because this is when the text interpreter is parsing for valid commands. It thus must skip over leading space characters. When parse is used to compile string literals, it will use the double quot character (ASCII 34) as the delimiting character. It the delimiting character is not space, parse starts scanning immediately, looking for the designated delimiting character.

parse	The elementary command to do text parsing. From the input stream, which starts at b1 and is of u1 characters long, it parses out the first text string delimited by character c. It returns the address b2 and length u2 of the string just parsed out and the difference n between b1 and b2. Leading delimiters are skipped over.
PARSE	Scan the input stream in the Terminal Input Buffer from where `>IN`

	points to, until the end of the buffer, for a string delimited by character c. It returns the address and length of the string parsed out. PARSE calls parse to do the detailed works. PARSE is used to implement many specialized parsing commands to perform different parsing functions.
.(	Print the following string till the next) character. It is used to output text to the serial output device.
(	Discard the following string till the next) character. It is used to place comments in source code.
\	Discard all characters till end of a line. It is used to insert comment lines in source code.
CHAR	Parse the next string out but returns only the first character in this string. It gets an ASCII character from the input stream.
TOKEN	Parse out the next string delimited by the space character. It then copies this string as a counted string to the first free area in RAM memory and returns its address. The length of the string is limited to 31 characters.
WORD	Parse out the next string delimited by the ASCII character c. It then copies this string as a counted string to the first free area in RAM memory and returns its address. The length of the string is limited to 255 characters.

```
;; Parsing

;  parse ( b u c -- b u delta ; <string> )
;  Scan string delimited by c. Return found string and its
offset.
    .word  QUEST-2
    .byte  5,"parse"
PARS
    call   #TEMP
    call   #STORE
    call   #OVER
    call   #TOR
    call   #DUPP
    call   #QBRAN
    .word  PARS8
    sub    #1,tos
    call   #TEMP
    call   #AT
    call   #BLANK
    call   #EQUAL
    call   #QBRAN
    .word  PARS3
```

```
    call   #TOR
PARS1: call   #BLANK
    call   #OVER
    call   #CAT   ;skip leading blanks ONLY
    call   #SUBB
    call   #ZLESS
    call   #INVER
    call   #QBRAN
    .word  PARS2
    add    #1,tos
    call   #DONXT
    .word  PARS1
    call   #RFROM
    call   #DROP,
    call   #DOLIT
    .word  0
    call   #DUPP
    ret
PARS2: call   #RFROM
PARS3: call   #OVER
    call   #SWAP
    call   #TOR
PARS4: call   #TEMP
    call   #AT
    call   #OVER
    call   #CAT
    call   #SUBB  ;scan for delimiter
    call   #TEMP
    call   #AT
    call   #BLANK
    call   #EQUAL
    call   #QBRAN
    .word  PARS5
    call   #ZLESS
PARS5:
    call   #QBRAN
    .word  PARS6
    add    #1,tos
```

```
    call   #DONXT
    .word  PARS4
    call   #DUPP
    call   #TOR
    call   #BRAN
    .word  PARS7
PARS6: call   #RFROM
    call   #DROP
    call   #DUPP
    add    #1,tos
    call   #TOR
PARS7: call   #OVER
    call   #SUBB
    call   #RFROM
    call   #RFROM
    call   #SUBB
    ret
PARS8: call   #OVER
    call   #RFROM
    call   #SUBB
    ret

;   PARSE  ( c -- b u ; <string> )
;   Scan input stream and return counted string delimited by
c.
    .word  PARS-6
    .byte  5,"PARSE"
PARSE
    call   #TOR
    call   #TIB
    call   #INN
    call   #AT
    call   #PLUS  ;current input buffer pointer
    call   #NTIB
    call   #AT
    call   #INN
    call   #AT
    call   #SUBB  ;remaining count
```

```
   call   #RFROM
   call   #PARS
   call   #INN
   call   #PSTOR
   ret

;  .( ( -- )
;  Output following string up to next ) .
   .word  PARSE-6
   .byte  IMEDD+2,".(",0
DOTPR
   call   #DOLIT
   .word  ")"
     Call#PARSE
   call   #TYPEE
   ret

;  (  ( -- )
;  Ignore following string up to next ) . A comment.
   .word  DOTPR-4
   .byte  IMEDD+1,"("
PAREN
   call   #DOLIT
   .word  ")"
     Call#PARSE
   call   #DDROP
   ret

;  \  ( -- )
;  Ignore following text till the end of line.
   .word  PAREN-2
   .byte  IMEDD+1,"\"
BKSLA
   call   #NTIB
   call   #AT
   call   #INN
   call   #STORE
   ret
```

```
;   CHAR   ( -- c )
;   Parse next word and return its first character.
    .word  BKSLA-2
    .byte  4,"CHAR",0
CHAR
    call   #BLANK
    call   #PARSE
    call   #DROP
    call   #CAT
    ret

;   TOKEN  ( -- a ; <string> )
;   Parse a word from input stream and copy it to name
dictionary.
    .word  CHAR-6
    .byte  5,"TOKEN"
TOKEN
    call   #BLANK
    call   #PARSE
    call   #DOLIT
    .word  31
    call   #MIN
TOKEN1
    call   #HERE
    call   #DDUP
    call   #CSTOR
    add    #1,tos
    call   #SWAP
    call   #CMOVE
    jmp    HERE

;   WORD   ( c -- a ; <string> )
;   Parse a word from input stream and copy it to code
dictionary.
    .word  TOKEN-6
    .byte  4,"WORD",0
WORDD
```

```
call   #PARSE
jmp    TOKEN1
```

4.10 Dictionary Search

In 430eForth, command records are linearly linked into a dictionary. A command record contains three fields: a link field holding the name field address of the previous command record, a name field holding the name as a counted string, and a code field holding executable code and data. A dictionary search follows the linked list of records to find a name which matches a text string. It returns the name field address and the code field address, if a match is found.

The link field of the first command record contains a 0, indicating it is the end of the linked list. A system variable `CONTEXT` holds an address pointing to the name field of the last command record. The dictionary search starts at `CONTEXT` and terminates at the first matched name, or at the first command record.

From `CONTEXT`, we locate the name field of the last command record in the dictionary. It this name does not match the string to be searched, we can find the link field of this record, which is 2 bytes less than the name field address. From the link field, we locate the name field of the next command record. Compare the name with the search string. And so forth.

NAME>	Convert a name field address in a command record to the code field address of this command record. Code field address is the name field address plus length of name plus one, and aligned to the next cell boundary.
SAME?	Compare two strings at addresses `a` and `b` for `u` bytes. It returns a 0 if two strings are equal. It returns a positive integer if `a` string is greater than `b` string. It returns a negative integer if `a` string is less than `b` string.
NAME?	Search the dictionary starting at `CONTEXT` for a name string at address `a`. Return the code field address and name field address if a matched command is found. Otherwise, return the original string address `a` and a false flag. Assume that a count string is at memory address `a`, and the name field address of the last command record is in address `va`. If the string matches the name of a command, both the code field address and the name field address of the command record are returned. If the string is not a valid command, the original string address and a false flag are returned. It runs the dictionary search very quickly because it first compares the length byte and the first character in the name field as a 16 bit integer. In most cases of mismatch, this comparison would fail and the next record can be reached through the link field. If the first two characters match, then `SAME?` is invoked to compare the rest of the name field, one cell at a time. Since both the target text string and the name field are null filled to the cell boundary, the comparison can be performed quickly across the entire name field without worrying about the end conditions.

```
;; Dictionary search

;   NAME>  ( na -- ca )
;   Return a code address given a name address.
    .word  WORDD-6
    .byte  5,"NAME>"
NAMET
    call   #COUNT
    and    #1FH,tos
    call   #PLUS
    jmp    ALGND

;   SAME?  ( a a u -- a a f \ -0+ )
;   Compare u cells in two strings. Return 0 if identical.
    .word  NAMET-6
    .byte  5,"SAME?"
SAMEQ
    call   #OVER
    call   #CAT
SAME1:
    mov 2(stack),temp0
    add tos,temp0
    mov.b  0(temp0),temp0
    mov 0(stack),temp1
    add tos,temp1
    mov.b  0(temp1),temp1
    sub temp1,temp0
    jnz SAME2
    dec tos
    jnz SAME1
    ret
SAME2:
    mov #-1,tos
    ret

;   NAME?  ( a -- ca na | a F )
;   Search all context vocabularies for a string.
```

```
    .word  SAMEQ-6
    .byte  5,"NAME?"
NAMEQ
    call   #CNTXT
    call   #AT
FIND1:
    tst    tos
    jz     FIND3     ;end of dictionary
    call   #OVER
    call   #AT
    call   #OVER
    call   #AT
    call   #DOLIT
    .word  MASKK
    call   #ANDD
    call   #EQUAL
    call   #QBRAN
    .word  FIND4
    call   #SAMEQ
    call   #QBRAN
    .word  FIND2         ;match
FIND4
    decd   tos
    mov    0(tos),tos
    jmp    FIND1
FIND2
    mov     tos,0(stack)
    call   #NAMET
    br     #SWAP
FIND3:
    ret
```

4.11 Terminal Input

The text interpreter interprets source text received from an input device and stored in the Terminal Input Buffer. To process characters in the Terminal Input Buffer, we need special commands to deal with the special conditions of backspace character and carriage return: On top of stack, three special parameters are referenced in many commands: `bot` is the Beginning Of the Text input buffer, `eot` is the End Of the Text

input buffer, and `cur` points to the current character in the input buffer.

^H	Process back-space character (ASCII 8). It erases the last character entered, and decrement the character pointer `cur`. If `cur=bot`, do nothing because you cannot backup beyond beginning of input buffer.
TAP	Output a character `c` to terminal, store `c` in `cur`, and increment the character pointer `cur`, which points to the current character in the input buffer. `bot` and `eot` are also pointers pointing to the beginning and end of the input buffer.
kTAP	Process character `c`. `bot` is pointing at the beginning of the input buffer, and `eot` is pointing at the end. `cur` points to the current character in the input buffer. The character `c` is normally stored at `cur`, which is then incremented by 1. If `c` is a carriage-return (ASCII 13), echo a space and make `eot=cur`., thus terminating the input process If `c` is a back-space (ASCII 8), erase the last character and decrement `cur`.
accept	Accept `u` characters into an input buffer starting at address `b`, or until a carriage return (ASCII 13) is encountered. The value of `u` returned is the actual number of characters received.
QUERY	Accept up to 80 characters from the input device to the Terminal Input Buffer. It also prepares the Terminal Input Buffer for parsing by setting `#TIB` to the length of the input text stream, and clearing `>IN` which points to the beginning of the Terminal Input Buffer.

```
;; Terminal response

;  ^H ( bot eot cur -- bot eot cur )
;  Backup the cursor by one character.
   .word  NAMEQ-6
   .byte  2,"^H",0
BKSP
   call   #TOR
   call   #OVER
   call   #RFROM
   call   #SWAP
   call   #OVER
   call   #XORR
   call   #QBRAN
   .word  BACK1
   call   #DOLIT
   .word  BKSPP
   call   #EMIT
   sub    #1,tos
```

```
    call   #BLANK
    call   #EMIT
    call   #DOLIT
    .word  BKSPP
    call   #EMIT
BACK1: ret

;   TAP( bot eot cur c -- bot eot cur )
;   Accept and echo the key stroke and bump the cursor.
    .word  BKSP-4
    .byte  3,"TAP"
TAP
    call   #DUPP
    call   #EMIT
    call   #OVER
    call   #CSTOR,
    add    #1,tos
    ret

;   kTAP   ( bot eot cur c -- bot eot cur )
;   Process a key stroke, CR or backspace.
    .word  TAP-4
    .byte  4,"kTAP",0
KTAP
    call   #DUPP
    sub    #CRR,tos
    call   #QBRAN
    .word  KTAP2
    sub    #BKSPP,tos
    call   #QBRAN
    .word  KTAP1
    call   #BLANK
    jmp    TAP
KTAP1:
    jmp    BKSP
KTAP2:
    call   #DROP
    call   #SWAP
```

```
    call   #DROP
    jmp    DUPP

;   accept ( b u -- b u )
;   Accept characters to input buffer. Return with actual
count.
    .word  KTAP-6
    .byte  6,"accept",0
ACCEP
    call   #OVER
    call   #PLUS
    call   #OVER
ACCP1: call   #DDUP
    call   #XORR
    call   #QBRAN
    .word  ACCP4
    call   #KEY
    call   #DUPP
    call   #BLANK
    call   #SUBB
    call   #DOLIT
    .word  95
    call   #ULESS
    call   #QBRAN
    .word  ACCP2
    call   #TAP
    call   #BRAN
    .word  ACCP1
ACCP2: call   #KTAP
ACCP3:
    jmp    ACCP1
ACCP4: call   #DROP
    call   #OVER
    jmp    SUBB

;   QUERY  ( -- )
;   Accept input stream to terminal input buffer.
    .word  ACCEP-8
```

```
    .byte  5,"QUERY"
QUERY
    call   #TIB,
    call   #DOLIT
    .word  80
    call   #ACCEP
    call   #NTIB
    call   #STORE
    call   #DROP
    call   #DOLIT
    .word  0
    call   #INN
    call   #STORE
    ret
```

4.12 Interpreter

Error Handling

When error occurred, it is usually because the text interpreter encounters a string which can not be interpreted or processed. This string is usually stored in a buffer in RAM memory.

ERROR	Print the string in RAM memory located at address `a`, followed by a `?` mark and aborts. 'Abort' means flushing all flash memory buffers, clearing the parameter stack, and returns to the text interpreter loop `QUIT`.
abort"	It is compiled with an error message string in a compound command. When `abort"` is executed, it examines the top item on the parameter stack. It the flag is true, print out the following error message and `QUIT`; otherwise, skip over the error message and continue execution the next token.

```
;; Error handling

;   ERROR  ( a -- )
;   Return address of a null string with zero count.

    .word  QUERY-6
    .byte  5,"ERROR"
ERROR:
```

```
    call   #SPACE
    call   #COUNT
    call   #TYPEE
    call   #DOLIT
    .word  3FH
    call   #EMIT
    call   #CR
;   call   #EMPTY_BUF
    jmp    QUIT

;   abort" ( f -- )
;   Run time routine of ABORT" . Abort with a message.
    .word  ERROR-6
    .byte  COMPO+6,"abort"""
ABORQ
    call   #QBRAN
    .word  ABOR1  ;text flag
    call   #DOSTR
    call   #COUNT
    call   #TYPEE
    jmp    QUIT   ;pass error string
ABOR1: call   #DOSTR
    call   #DROP
    ret ;drop error
```

Interpreter

Text interpreter in FORTH is like a conventional operating system of a computer. It is the primary interface a user uses to get the computer to do work. Since FORTH uses very simple syntax rule--commands are separated by spaces, the text interpreter is also very simple. It accepts a line of text from the terminal, parses out a command delimited by spaces, locates the command in the dictionary and then executes it. The process is repeated until the input text is exhausted. Then the text interpreter waits for another line of text and interprets it again. This cycle repeats until you are exhausted and turns off the computer.

In 430eForth, the text interpreter is coded as the command `QUIT`. `QUIT` contains an infinite loop which repeats the `QUERY-EVAL` command pair. `QUERY` accepts a line of text from the input terminal. `EVAL` interprets the text one command at a time till the end of the text line.

$INTERPRET	Execute a command whose name string is stored at address a on the

	parameter stack. If the string is not a valid command, convert it to a number. Failing the numeric conversion, execute ERROR and return to QUIT.
[	Activate the text interpreter by storing the code field address of $INTERPRET into the variable 'EVAL, which is executed in EVAL while the text interpreter is in the interpretive mode.
.OK	Print the familiar ok> prompting message after executing to the end of a line. The message ok> is printed only when the text interpreter is in the interpretive mode. While compiling, the prompt is suppressed.
?STACK	Check for stack underflow. Abort, resetting the parameter stack pointer, if the stack depth is negative.
EVAL	It is contained in the text interpreter loop which parses commands from the input stream and invokes whatever token in 'EVAL to process the commands, either execute it with $INTERPRET or compile it with $COMPILE.
QUIT	It is the operating system, the text interpreter, or a shell, of the 430eForth system. It is an infinite loop eForth will never get out. It uses QUERY to accept a line of commands from the input terminal and then lets EVAL to parse out the commands and execute them. After a line is processed, it displays an ok> message and wait for the next line of commands. When an error occurred during execution, it prints the string which caused the error as an error message. After the error is reported, it re-initializes the system by clearing the return stack and comes back to receive the next line of commands. Because the behavior of EVAL can be changed by storing either $INTERPRET or $COMPILE into 'EVAL, QUIT exhibits the dual nature of a text interpreter and a compiler.

```
;; The text interpreter

;   $INTERPRET   ( a -- )
;  Interpret a word. If failed, try to convert it to an
integer.
    .word  ABORQ-8
    .byte  10,"$INTERPRET",0
INTER
    call   #NAMEQ
    call   #QDUP  ;?defined
    call   #QBRAN
    .word  INTE1
    call   #AT
    call   #DOLIT
```

```
    .word  COMPO
    call   #ANDD  ;?compile only lexicon bits
    call   #ABORQ
    .byte  13," compile only"
    call   #EXECU
    ret ;execute defined word
INTE1: call   #NUMBQ
    call   #QBRAN
    .word  INTE2
    ret
INTE2: jmp ERROR     ;error

;   [  ( -- )
;   Start the text interpreter.
    .word  INTER-12
    .byte  IMEDD+1,"["
LBRAC
    call   #DOLIT
    .word  INTER
    call   #TEVAL
    call   #STORE
    ret

;   .OK( -- )
;   Display 'ok' only while interpreting.
    .word  LBRAC-2
    .byte  3,".OK"
DOTOK
    call   #DOLIT
    .word  INTER
    call   #TEVAL
    call   #AT
    call   #EQUAL
    call   #QBRAN
    .word  DOTO1
    call   #DOTQP
    .byte  3," ok"
DOTO1: call   #CR
```

```
    Ret

;   ?STACK ( -- )
;   Abort if the data stack underflows.
    .word  DOTOK-4
    .byte  6,"?STACK",0
QSTAC
    call   #DEPTH
    call   #ZLESS ;check only for underflow
    call   #ABORQ
    .byte  10," underflow",0
    Ret

;   EVAL   ( -- )
;   Interpret the input stream.
    .word  QSTAC-8
    .byte  4,"EVAL",0
EVAL
EVAL1: call   #TOKEN
    call   #DUPP
    call   #CAT   ;?input stream empty
    call   #QBRAN
    .word  EVAL2
    call   #TEVAL
    call   #ATEXE
    call   #QSTAC ;evaluate input, check stack
    call   #BRAN
    .word  EVAL1
EVAL2: call   #DROP
    call   #DOTOK
    ret ;prompt

;; Shell

;   QUIT   ( -- )
;   Reset return stack pointer and start text interpreter.
    .word  EVAL-6
    .byte  4,"QUIT",0
```

```
QUIT
    mov #SPP,stack
    mov #RPP,SP
QUIT1: call   #LBRAC    ;start interpretation
QUIT2: call   #QUERY    ;get input
    call   #EVAL
    jmp    QUIT2  ;continue till error
```

4.13 Compiler

In MSP430G2553, the flash main memory is organized in 512 byte pages, and the flash information memory is organized in 64 byte pages. The flash memory can be read like RAM memory, but to write and to erase flash memory, you have to go through the flash memory controller. The flash memory controller makes everything very easy. You first unlock the memory controller, and then issue a write or erase command. Then you write to one location in the flash memory page, and finally lock the flash memory controller. Following are the commands to write one 16-bit integer to a flash memory location, to erase one page of flash memory, and to copy an array from a memory area to a flash memory area.

'	Search the dictionary for the following string. If the string is a valid command, return its code field address. If the string is not a valid command, print a ? mark.
ALLOT	Allocate `n` bytes of RAM memory on bottom of the free RAM space. System variable DP points to the bottom of free RAM space.
IALLOT	Allocate `n` bytes of flash memory on the top of the dictionary. System variable CP points to the top of the dictionary.
I!	Store the 16-bit data w in flash memory address `a`.
ERASE	Erase one 512 byte page of flash main memory or 64 bytes of flash information memory. The page address a is on the top of the parameter stack.
WRITE	Copy `n` bytes of one memory array, starting at address `src`, to an array in flash memory, starting at flash address `dest`. All addresses are byte addresses.

The command I! is actually the primitive compiler in eForth. It allows us to write into the flash memory to build new FORTH commands. It is used to define the ‘,’ (comma) command, which add one more 16-bit integer to the top of the FORTH dictionary, and thus extends the FORTH system by one integer. Repeatedly adding data and instructions to the dictionary to form new FORTH commands is what a compiler does.

```
;; The compiler

;   '  ( -- ca )
;  Search context vocabularies for the next word in input
stream.
    .word  QUIT-6
    .byte  1,"'"
TICK
    call   #TOKEN
    call   #NAMEQ ;?defined
    call   #QBRAN
    .word  TICK1
    ret    ;yes, push code address
TICK1: jmp ERROR     ;no, error

;   ALLOT  ( n -- )
;  Allocate n bytes to the RAM dictionary.
    .word  TICK-2
    .byte  5,"ALLOT"
ALLOT
    call   #DP
    jmp    PSTOR

;   IALLOT ( n -- )
;  Allocate n bytes to the code dictionary.
    .word  ALLOT-6
    .byte  6,"IALLOT",0
IALLOT
    call   #CP
    jmp    PSTOR

;   I! ( n a -- )
;  Store n to address a in code dictionary.
    .word  IALLOT-8
    .byte  2,"I!",0
ISTORE
    mov    #FWKEY,&FCTL3 ; Clear LOCK
    mov    #FWKEY+WRT,&FCTL1 ; Enable write
```

```
    call   #STORE
    mov    #FWKEY,&FCTL1 ; Done. Clear WRT
    mov    #FWKEY+LOCK,&FCTL3 ; Set LOCK
    ret

;   ERASE  ( a -- )
;   Erase a segment at address a.
    .word  ISTORE-4
    .byte  5,"ERASE"
IERASE
    mov    #FWKEY,&FCTL3 ; Clear LOCK
    mov    #FWKEY+ERASE,&FCTL1 ; Enable erase
    clr    0(tos)
    mov    #FWKEY+LOCK,&FCTL3 ; Set LOCK
    loadtos
    ret

;   WRITE  ( src dest n -- )
;   Copy n byte from src to dest.  Dest is in flash memory.
    .word  IERASE-6
    .byte  5,"WRITE"
WRITE
    rra    tos
    call   #TOR
WRITE1
    call   #OVER
    call   #AT
    call   #OVER
    call   #ISTORE
    incd   tos
    incd   0(stack)
    call   #DONXT
    .word  WRITE1
    jmp    DDROP

    jmp    COMMA
```

Compiler Commands

,	It is the most fundamental compiler command. It compiles an integer w to dictionary in the flash memory, and add the new item to the growing command list of the current command under construction. This is the primitive compiler upon which the FORTH compiler rests.
CALL,	Compile or assemble a subroutine call instruction with the code field address on the parameter stack as destination. Compound commands are compiled as lists of subroutine calls.
[COMPILE]	Compile the code field address of the next command in the input stream. It is used to compile commands, which would otherwise be executed while compiling.
COMPILE	Compile the code field address of the next command in the input stream. It forces compilation of a command at run time.
LITERAL	Compile an integer literal. It first compiles a call doLIT machine instruction, followed by an integer value from the parameter stack. When doLIT is executed, it extracts the integer in the next program word and pushes it on the parameter stack.
$,"	Compile a string literal. String text is taken from the input stream and terminated by a double quote. A token (such as . "| or $"|) must be compiled before the string to form a sting literal.

```
;   ,   ( w -- )
;   Compile an integer into the code dictionary.
    .word  WRITE-6
    .byte  1,","
COMMA
    call   #CP
    CALL   #AT
    call   #DUPP
    call   #CELLP ;cell boundary
    call   #CP
    call   #STORE
    jmp    ISTORE

;   call,  ( w -- )
;   Compile a call instruction into the code dictionary.
    .word  COMMA-2
    .byte  5,"call,"
CALLC
    call   #DOLIT
    .word  CALLL
    call   #COMMA
```

```
;   [COMPILE] ( -- ; <string> )
;   Compile the next immediate word into code dictionary.
    .word  CALLC-6
    .byte  IMEDD+9,"[COMPILE]"
BCOMP
    call   #TICK
    jmp    CALLC

;   COMPILE   ( -- )
;   Compile the next address in colon list to code dictionary.
    .word  BCOMP-10
    .byte  COMPO+7,"COMPILE"
COMPI
    call   #RFROM
    call   #DUPP
    call   #AT
    call   #COMMA ;compile call instruction
    call   #CELLP
    call   #DUPP
    call   #AT
    call   #COMMA ;compile address
    call   #CELLP
    call   #TOR
    ret           ;adjust return address

;   LITERAL   ( w -- )
;   Compile tos to code dictionary as an integer literal.
    .word  COMPI-8
    .byte  IMEDD+7,"LITERAL"
LITER
    call   #DOLIT
    .word  DOLIT
    call   #CALLC
    jmp    COMMA

;   $,"( -- )
;   Compile a literal string up to next " .
    .word  LITER-8
```

```
    .byte  3,"$,"""
STRCQ
    call   #DOLIT
    .word  """"
    call   #WORDD ;move string to code dictionary
STRCQ1
    call   #DUPP
    call   #CAT
    call   #TWOSL ;calculate aligned end of string
    call   #TOR
STRCQ2
    call   #DUPP
    call   #AT
    call   #COMMA
    call   #CELLP
    call   #DONXT
    .word  STRCQ2
    jmp    DROP
```

Structure Commands

Immediate commands are not compiled as tokens by the compiler. Instead, they are executed by the compiler immediately. They are used to build control structures in compound commands. Immediate commands has its IMMEDIATE lexicon bit set, in the length byte of the name field. The control structures used in 430eForth are the following:

Conditional branch	`IF ... THEN`
	`IF ... ELSE ... THEN`
Finite loop	`FOR ... NEXT`
	`FOR ... AFT ... THEN... NEXT`
Infinite loop	`BEGIN ... AGAIN`
Indefinite loop	`BEGIN ... UNTIL`
	`BEGIN ... WHILE ... REPEAT`

A control structure contains one or more address literals with ?branch, branch and next commands, which causes execution to branch out of the normal sequence. The control structure commands are immediate commands which compile the address literals and resolve the branch address.

One should note that `BEGIN` and `THEN` do not compile any token. They set up or resolve control structures in compound commands. `IF`, `ELSE`, `WHILE`, `UNTIL`, and `AGAIN` do compile address literals with branching tokens.

I use two characters a and A to denote some addresses on the data stack. a points to a location to where a branch commands would jump to. A points to a location where a new address will be stored when the address is resolved.

BEGIN	Start a loop structure. It pushes an address a on the parameter stack. a points to the top of the dictionary where new tokens will be compiled. If begins an infinite loop or an indefinite loop.
FOR	Compile a >R token and pushes the address of the next token a on the parameter stack. It starts a FOR-NEXT loop.
NEXT	Compile a next token with a target address a on the top of the parameter stack. It resolves a FOR NEXT loop.
UNTIL	Compile a ?branch token with a target address a on the top of the parameter stack. It resolves a BEGIN-UNTIL loop.
AGAIN	Compile a branch token with a target address a on the top of the parameter stack. It resolves a BEGIN-AGAIN loop.
IF	Compile a ?branch address literal and pushes its address, a, is left on the parameter stack. It starts a IF-ELSE-THEN or a IF-THEN branch structure.
AHEAD	Compile a branch address literal and pushes its address, a, is left on the parameter stack. It starts a AHEAD-THEN branch structure.
REPEAT	Compile a branch token with a target address a on the top of the parameter stack. It resolves a BEGIN-WHILE-REPEAT loop.
THEN	Resolve the address in a branch token whose address is a on the top of the parameter stack. It resolves a IF-ELSE-TEHN or IF-THEN branch structure.
AFT	Compile a branch literal and leaves its address as A, It also replaces the address a left by FOR with the address a1 of the next token. A will be used by THEN to resolve the AFT-THEN branch structure, and a1 will be used by NEXT to resolve the loop structure.
ELSE	Compile a branch token, and use the address of the next token to resolve the address field of ?branch token in a, as left by IF. It also replaces a with A, the address of its address field for THEN to resolve. ELSE starts the false clause in the IF-ELSE-THEN branch structure.
WHILE	Compile a ?branch token and leave its address, A, on the stack. Address a left by BEGIN is swapped to the top of the parameter stack. WHILE is used to start the true clause in the BEGIN-WHILE-REPEAT loop.

```
;; Structures

;   FOR( -- a )
;   Start a FOR-NEXT loop structure in a colon definition.
```

```
    .word  STRCQ-4
    .byte  IMEDD+3,"FOR"
FOR
    call   #DOLIT
    .word  TOR
    call   #CALLC
    jmp    BEGIN

;   BEGIN  ( -- a )
;   Start an infinite or indefinite loop structure.
    .word  FOR-4
    .byte  IMEDD+5,"BEGIN"
BEGIN
    call   #CP
    jmp    AT

;   NEXT   ( a -- )
;   Terminate a FOR-NEXT loop structure.
    .word  BEGIN-6
    .byte  IMEDD+4,"NEXT",0
NEXT
    call   #DOLIT
    .word  DONXT
    call   #CALLC
    jmp    COMMA

;   UNTIL  ( a -- )
;   Terminate a BEGIN-UNTIL indefinite loop structure.
    .word  NEXT-6
    .byte  IMEDD+5,"UNTIL"
UNTIL
    call   #DOLIT
    .word  QBRAN
    call   #CALLC
    jmp    COMMA

;   AGAIN  ( a -- )
;   Terminate a BEGIN-AGAIN infinite loop structure.
```

```
    .word  UNTIL-6
    .byte  IMEDD+5,"AGAIN"
AGAIN
    call   #DOLIT
    .word  BRAN
    call   #CALLC
    jmp    COMMA

;   IF ( -- A )
;   Begin a conditional branch structure.
    .word  AGAIN-6
    .byte  IMEDD+2,"IF",0
IFF
    call   #DOLIT
    .word  QBRAN
    call   #CALLC
    call   #BEGIN
    call   #DOLIT
    .word  2
    jmp    IALLOT

;   AHEAD  ( -- A )
;   Compile a forward branch instruction.
    .word  IFF-4
    .byte  IMEDD+5,"AHEAD"
AHEAD
    call   #DOLIT
    .word  BRAN
    call   #CALLC
    call   #BEGIN
    call   #DOLIT
    .word  2
    jmp    IALLOT

;   REPEAT ( A a -- )
;   Terminate a BEGIN-WHILE-REPEAT indefinite loop.
    .word  AHEAD-6
    .byte  IMEDD+6,"REPEAT",0
```

```
REPEA
   call   #AGAIN
   call   #BEGIN
   call   #SWAP
   jmp    ISTORE

;  THEN   ( A -- )
;  Terminate a conditional branch structure.
   .word  REPEA-8
   .byte  IMEDD+4,"THEN",0
THENN
   call   #BEGIN
   call   #SWAP
   jmp    ISTORE

;  AFT( a -- a A )
;  Jump to THEN in a FOR-AFT-THEN-NEXT loop the first time
through.
   .word  THENN-6
   .byte  IMEDD+3,"AFT"
AFT
   call   #DROP
   call   #AHEAD
   call   #BEGIN
   jmp    SWAP

;  ELSE   ( A -- A )
;  Start the false clause in an IF-ELSE-THEN structure.
   .word  AFT-4
   .byte  IMEDD+4,"ELSE",0
ELSEE
   call   #AHEAD
   call   #SWAP
   jmp    THENN

;  WHILE  ( a -- A a )
;  Conditional branch out of a BEGIN-WHILE-REPEAT loop.
   .word  ELSEE-6
```

```
    .byte  IMEDD+5,"WHILE"
WHILE
    call   #IFF
    jmp    SWAP
```

ABORT"	Compile an error message as a string literal. This error message is display at run time if the top item on the parameter stack is `true`, and the rest of the tokens in this compound command are skipped and eForth enters the interpreter loop in `QUIT`. This is the programmed response to an error condition.
."	Compile a string literal which will be printed when it is executed in run time. This is the best way to present messages to you in an application.
$"	Compile a string literal. When it is executed, only the address of the string is pushed on the parameter stack. Later commands can use this address to access the string and individual characters in the string as a string array.

```
;   ABORT" ( -- ; <string> )
;   Conditional abort with an error message.
    .word  WHILE-6
    .byte  IMEDD+6,"ABORT""",0
ABRTQ
    call   #DOLIT
    .word  ABORQ
    call   #CALLC
    jmp    STRCQ

;   $" ( -- ; <string> )
;   Compile an inline string literal.
    .word  ABRTQ-8
    .byte  IMEDD+2,"$""",0
STRQ
    call   #DOLIT
    .word  STRQP
    call   #CALLC
    call   #STRCQ
    ret

;   ." ( -- ; <string> )
;   Compile an inline string literal to be typed out at run
```

```
time.
    .word  STRQ-4
    .byte  IMEDD+2,".""",0
DOTQ
    call   #DOLIT
    .word  DOTQP
    call   #CALLC
    call   #STRCQ
    ret
```

Name Compiler

We had seen how tokens and structures are compiled into the code field of a compound command in the dictionary. To build a new command, we have to build its header first. A header consists of a link field and a name field. Here are the commands to build the header.

?UNIQUE	Display a warning message to show that the name of a new command already exists in the dictionary. FORTH does not prevent your reusing the same name for different commands. However, giving the same name to many different commands often causes problems in software projects. It is to be avoided if possible and `?UNIQUE` reminds you of it.
$,n	Build a new header with a name string at RAM address `na`. It first build a link field with an address pointing to the name field of the prior command, and then copies the string at `na` to build a name field. The top of dictionary is the code field of the new command, and tokens can be compiled.

```
;; Name compiler

;   ?UNIQUE   ( a -- a )
;   Display a warning message if the word already exists.
    .word  DOTQ-4
    .byte  7,"?UNIQUE"
UNIQU
    call   #DUPP
    call   #NAMEQ ;?name exists
    call   #QBRAN
    .word  UNIQ1  ;redefinitions are OK
    call   #DOTQP
    .byte  7," reDef "   ;but warn the user
```

```
    call   #OVER
    call   #COUNT
    call   #TYPEE ;just in case its not planned
UNIQ1: jmp    DROP

;   $,n( na -- )
;   Build a new dictionary name using the string at na.
    .word  UNIQU-8
    .byte  3,"$,n"
SNAME
    call   #DUPP
    call   #CAT      ;?null input
    call   #QBRAN
    .word  SNAM1
    call   #UNIQU    ;?redefinition
    call   #LAST
    call   #AT
    call   #COMMA    ;save na for vocabulary link
    call   #CP
    call   #AT
    call   #LAST
    call   #STORE
    jmp    STRCQ1    ;fill name field
SNAM1
    call   #STRQP
    .byte  5," name" ;null input
    jmp    ERROR
```

FORTH Compiler

$COMPILE	Build the token list of a new compound command in its code field, which is on the top of the dictionary. It takes a string address a on the top of the parameter stack, search dictionary for a matching command, and adds a token to the token list. If the string is not a valid command, it is converted to a number, and a integer literal added to the token list. If the string is not a number, abort the compilation process and return to the text interpreter loop in QUIT. If the string is the name of an immediate command, this command is not compiled, but executed immediately. Immediate commands are tools used by the compiler to build structures in compound commands.

OVERT	Link a new command to the dictionary and thus makes it available for dictionary searches. When a new header is build, its name field address is stored in system variable LAST, and it is not yet linked to the dictionary which starts at CONTEXT. OVERT copies the name field address in LAST to CONTEXT and links the new command to the dictionary. It is used to protect the dictionary so that new commands not compiled successfully will not be compiled incorrectly into later compound commands.
;	Terminate a new compound command. It compiles an ret machine instruction to terminate the new token list, links this new command to the dictionary, and then returns to the text interpreter by storing the code field address of $INTERPRET into system variable 'EVAL.
]	Turn the text interpreter to a compiler by storing the code field address of $COMPILE into system variable 'EVAL.
:	Create a new header and start a new compound command. It takes the following string in the input stream to be the name of the new command. The dictionary is ready to accept a token list.] turns the text interpreter into compiler, which will compile the following text strings to build a new compound command. The new compound command is terminated by ;.
IMMEDIATE	Set the immediate lexicon bit in the name field of the new command. When the compiler encounters a command with this bit set, it will not compile this word into the token list under construction, but execute it immediately. This bit allows structure commands to build special structures in compound commands, and to deal with special conditions when the compiler is running.

```
;; FORTH compiler

;   $COMPILE  ( a -- )
;   Compile next word to code dictionary as a token or literal.
    .word  SNAME-4
    .byte  8,"$COMPILE",0
SCOMP
    call   #NAMEQ
    call   #QDUP  ;?defined
    call   #QBRAN
    .word  SCOM2
    call   #AT
    call   #DOLIT
    .word  IMEDD
    call   #ANDD  ;?immediate
    call   #QBRAN
    .word  SCOM1
```

```
   jmp    EXECU  ;its immediate, execute
SCOM1:
   jmp    CALLC  ;its not immediate, compile
SCOM2:
   call   #NUMBQ ;try to convert to number
   call   #QBRAN
   .word  SCOM3
   jmp    LITER  ;compile number as integer
SCOM3:
   jmp    ERROR  ;error

;  OVERT  ( -- )
;  Link a new word into the current vocabulary.
   .word  SCOMP-10
   .byte  5,"OVERT"
OVERT
   call   #LAST
   call   #AT
   call   #CNTXT
   jmp    STORE

;  ;  ( -- )
;  Terminate a colon definition.
   .word  OVERT-6
   .byte  IMEDD+COMPO+1,";"
SEMIS
   call   #DOLIT
   ret
   call   #COMMA
   call   #LBRAC
   jmp    OVERT

;  ]  ( -- )
;  Start compiling the words in the input stream.
   .word  SEMIS-2
   .byte  1,"]"
RBRAC
   call   #DOLIT
```

```
    .word  SCOMP
    call   #TEVAL
    jmp    STORE

;   :  ( -- ; <string> )
;   Start a new colon definition using next word as its name.
    .word  RBRAC-2
    .byte  1,":"
COLON
    call   #TOKEN
    call   #SNAME
    jmp    RBRAC

;   IMMEDIATE ( -- )
;   Make the last compiled word an immediate word.
    .word  COLON-2
    .byte  9,"IMMEDIATE"
IMMED
    call   #DOLIT
    .word  IMEDD
    call   #LAST
    call   #AT
    call   #AT
    call   #ORR
    call   #LAST
    call   #AT
    jmp    ISTORE
```

Defining Commands

Defining commands are molds which can be used to create classes of commands which share the same run time execution behavior. In 430eForth, we have these defining commands: `:` , `CREATE`, `CONSTANT` and `VARIABLE`.

doCON	Fetch a value stored after the `call doCON` instruction and pushes it on the parameter stack. It returns to its caller immediately. The `call doCON` instruction and the value after it forms the code field in all constant commands. For variables and data arrays, the value pointing to a location in RAM memory. All commands are defined in flash memory which can hold only constants. However, the constants

	can be pointers to variables and arrays in RAM memory.
CREATE	Create a new data array in RAM memory without allocating memory. When commands created by `CREATE` is executed, they will push their respective RAM addresses on the parameter stack. Memory space of an actual array is allocated using `ALLOT` command.
VARIABLE	Create a new command with a `doCON` token followed by a pointer to RAM memory and allocate 2 bytes of space in RAM memory. When a variable commands is executed, it pushes this RAM address on the parameter stack.
CONSTANT	Create a new command with a `doCON` token followed by the constant value. When a constant command is executed, it pushes the constant value on the parameter stack.

```
;; Defining words

;   doCON  ( -- a )
;   Run time routine forCONSTANT, VARIABLE and CREATE.
    .word  IMMED-10
    .byte  COMPO+5,"doCON"
DOCON:
    savetos
    pop    tos
    MOV    @tos,tos
    ret

;   HEADER ( -- ; <string> )
;   Compile a new array entry without allocating code space.
    .word  DOCON-6
    .byte  6,"HEADER",0
HEADER
    call   #TOKEN
    call   #SNAME
    call   #OVERT
    call   #DOLIT
    .word  DOCON
    jmp    CALLC

;   CREATE ( -- ; <string> )
;   Compile a new array entry without allocating code space.
    .word  HEADER-8
    .byte  6,"CREATE",0
```

```
CREAT
    call   #HEADER
    call   #DP
    call   #AT
    jmp    COMMA

;   CONSTANT  ( n -- ; <string> )
;   Compile a new constant.
    .word  CREAT-8
    .byte  8,"CONSTANT",0
CONST
    call   #HEADER
    jmp    COMMA

;   VARIABLE  ( -- ; <string> )
;   Compile a new variable initialized to 0.
    .word  CONST-10
    .byte  8,"VARIABLE",0
VARIA
    call   #CREAT
    call   #DOLIT
    .word  2
    jmp    ALLOT
```

4.14 Tools

430eForth is a very small system and only a very small set of tool commands are provided. Nevertheless, this set of tool commands is powerful enough to help you debug new commands he adds to the system. They are also very interesting programming examples on how to use the commands in eForth to build applications.

Generally, the tool commands present information stored in different parts of the CPU in appropriate formats to let you inspect the results as he executes commands in the eForth system and commands he defined himself. The tool commands include memory dump, stack dump, dictionary dump, etc.

One important discipline in learning FORTH is to learn how to use the parameter stack effectively. All commands must consume their input parameters on the stack and leave only their intended results on the stack. Sloppy usage of the parameter stack is often the cause of bugs which are very difficult to detect later, as unexpected items left on the stack could result in unpredictable behavior. `.S` should be used liberally during programming and debugging to ensure that the correct parameters are

left on the parameter stack.

The parameter stack is the center for arithmetic and logic operations. It is where commands receive their parameters and also where they left their results. In debugging a new command which may use stack items and leave items on the stack, the best was to debug it is to inspect the parameter stack, before and after its execution. To inspect the parameter stack non-destructively, use the command `.S`.

DUMP	Print 128 bytes of data starting at RAM address `b` to the terminal. It dumps 16 bytes to a line. A line begins with the address of the first byte, followed by 16 bytes shown in hex, 3 columns per bytes. At the end of a line are the 16 bytes shown in ASCII characters. Non-printable characters are replaced by underscores (ASCII 95). `DUMP` command in most FORTH system takes an address and a length as parameters to dump a memory array.
>NAME	Return a code field address, xt, of a command from its name field address, na. If xt is not a valid code field address, return 0. It follows the linked list of the dictionary, and from every name field address we can get a corresponding code field address. If this address is not the same as xt, we go to the name field of the next command. If xt is a valid code field address, we surely will find it. If the entire dictionary is searched and xt is not found, it is not a valid code field address.
.ID	Display the name of a command, given the name field address of this command. It replaces non-printable characters in a name by under-scores.
WORDS	Display all the names in the dictionary. The order of words is reversed from the compiled order. The last defined command is shown first.

```
;; Tools

;   DUMP   ( a u -- )
;   Dump u bytes from a, in a formatted manner.
    .word  VARIA-10
    .byte  4,"DUMP",0
DUMP
    call   #DOLIT
    .word  7
    call   #TOR      ;start count down loop
DUMP1: call   #CR,
    call   #DUPP
    call   #DOLIT
    .word  5
```

```
   call   #UDOTR
   call   #DOLIT
   .word  15
   call   #TOR
DUMP2
   call   #COUNT
   call   #DOLIT
   .word  3
   call   #UDOTR
   call   #DONXT
   .word  DUMP2  ;loop till done
   call   #SPACE
   call   #DUPP
   sub    #16,tos
   call   #DOLIT
   .word  16
   call   #TYPEE ;display printable characters
   call   #DONXT
   .word  DUMP1  ;loop till done
   jmp    DROP

;   .S ( ... -- ... )
;  Display the contents of the data stack.
   .word  DUMP-6
   .byte  2,".S",0
DOTS
   call   #CR
   call   #DEPTH ;stack depth
   call   #TOR      ;start count down loop
   jmp    DOTS2  ;skip first pass
DOTS1:
   call   #RAT
   call   #PICK
   call   #DOT   ;index stack, display contents
DOTS2:
   call   #DONXT
   .word  DOTS1  ;loop till done
   call   #DOTQP
```

```
    .byte  4," <sp",0
    Ret

;   >NAME  ( ca -- na | F )
;   Convert code address to a name address.
    .word  DOTS-4
    .byte  5,">NAME"
TNAME
    call   #TOR
    call   #CNTXT    ;vocabulary link
    call   #AT
TNAM1:
    call   #DUPP  ;check all vocabularies
    call   #QBRAN
    .word  TNAM2
    call   #DUPP
    call   #NAMET
    call   #RAT
    call   #XORR  ;compare
    call   #QBRAN
    .word  TNAM2
    call   #CELLM    ;continue with next word
    call   #AT
    call   #BRAN
    .word  TNAM1
TNAM2:
    call   #RFROM
    jmp    DROP

;   .ID( na -- )
;   Display the name at address.
    .word  TNAME-6
    .byte  3,".ID"
DOTID
    call   #COUNT
    call   #DOLIT
    .word  01FH
    call   #ANDD  ;mask lexicon bits
```

```
    jmp    TYPEE

;   WORDS  ( -- )
;   Display the names in the context vocabulary.
    .word  DOTID-4
    .byte  5,"WORDS"
WORDS
    call   #CR
    call   #CNTXT
    call   #AT ;only in context
WORS1:
    call   #QDUP     ;?at end of list
    call   #QBRAN
    .word  WORS2
    call   #DUPP
    call   #SPACE
    call   #DOTID ;display a name
    call   #CELLM
    call   #AT
    call   #BRAN
    .word  WORS1
WORS2: ret
```

5.14 FORTH Startup

The startup routine `main` is located at the beginning of the flash main memory, at location C000H. It initializes the return stack pointer in the `SP` register, and the parameter stack pointer in `stack` register. It thus completes hardware initialization, and then jumps to COLD command which initializes the 430eForth FORTH Virtual Machine, by copying the system variables from flash information memory Segment D (at 1000H) to RAM memory starting at 200H, and starts running an application. The default application in 430eForth is `hi`, which simply sends out a sign-on message and falls into the text interpreter QUIT. The address of `hi` is stored in memory location named 'BOOT at 200H. This address can be changed to point to an application command. To build a turnkey system, the system variables must be saved in Segment D so that when the application boots up, it has all the properly initialized system variables.

Because all the system variable in 430eForth are initialized from a data array in the information flash memory, 430eForth is eminently ROMable and suitable for embedded applications in MSP430G2553. Before falling into `QUIT` to enter into the

text interpreter loop, COLD command executes a boot routine whose code address is stored in system variable 'BOOT. This code address can be vectored to an application command which defines the proper behavior of the system on power-up and on reset. Initially 'BOOT contains the code field address of hi.

hi	The default start-up routine in 430eForth. It initializes the serial I/O device and then displays a sign-on message. This is where you can customize his application. From here one can initialize the system to start his own application.
'BOOT	A system variable loaded at RAM memory address $100. It is originally vectored to hi.
COLD	A high level compound command executed upon power-up, called from the low level START routine. Its initializes the system variables, executes the boot-up routine vectored through 'BOOT, and then falls into the text interpreter loop QUIT.

```
;; Hardware reset

;   hi ( -- )
;   Display the sign-on message of eForth.
    .word  WORDS-6
    .byte  2,"hi",0
HI
;   call   #STOIO
    call   #CR ;initialize I/O
    call   #DOTQP
    .byte  14,"430eForth v1.0",0   ;model
    jmp    CR

;   'BOOT  ( -- a )
;   The application startup vector.
    .word  HI-4
    .byte  5,"'BOOT"
TBOOT
    savetos
    mov #UPP,tos
    ret

;   COLD   ( -- )
;   The hilevel cold start sequence.
    .word  TBOOT-6
```

```
    .byte  4,"COLD"
COLD
COLD1:
    call   #DOLIT
    .word  UZERO
    call   #DOLIT
    .word  UPP
    call   #DOLIT
    .word  ULAST-UZERO
    call   #CMOVE ;initialize user area
    call   #TBOOT
    call   #ATEXE ;application boot
    call   #QUIT     ;start interpretation
    jmp    COLD1  ;just in case

CTOP   .word  0FFFFH    ;next available memory in code
dictionary
```

System Variables

The first 32 bytes starting at location $200 are used by system variables, as shown in the following list:

Variable	Address	Function
'BOOT	200H	Execution vector to start application command.
BASE	202H	Radix base for numeric conversion.
tmp	204H	Scratch pad.
HLD	206H	Pointer to a buffer holding next digit for numeric conversion.
>IN	208H	Input buffer character pointer used by text interpreter.
#TIB	20AH	Number of characters in input buffer.
'EVAL	20CH	Execution vector switching between $INTERPRET and $COMPILE.
CONTEXT	210H	Vocabulary array pointing to last name fields of dictionary.
CP	212H	Pointer to top of dictionary, the first available flash memory location to compile new command
DP	214H	Pointer to the first available RAM memory location.
LAST	216H	Pointer to name field of last command in dictionary.

The initial values of these variable are stored in the information flash memory Segment D at location 1000H. The assembly commands `.sect ".infoD"` direct

the linker to store these initial values there so that the `COLD` routine can initializes them properly after boot-up.

The assembly commands `.sect ".reset"` direct the linker to store the address of `main` in the reset vector at 0FFFEH, so that MSP430G2553 jumps to `main` on boot-up.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;
; COLD start moves the following to USER variables.
; MUST BE IN SAME ORDER AS USER VARIABLES.

   .sect  ".infoD"
UZERO:
   .word  HI     ;200H, boot routine
   .word  BASEE  ;202H, BASE
   .word  0      ;204H, tmp
   .word  0      ;206H, >IN
   .word  0      ;208H, #TIB
   .word  INTER  ;20AH, 'EVAL
   .word  0      ;20CH, HLD
   .word  COLD-6 ;20EH, CONTEXT pointer
   .word  CTOP   ;210H, CP
   .word  DPP    ;220H, DP
   .word  COLD-6 ;214H, LAST
ULAST:

;=========================================================
=======
   .sect   ".reset"             ; MSP430 RESET Vector
   .short  main                ;  .end
;=========================================================
=======
```

Conclusions

What I give you in 430eForth is that in about 6000 bytes, you have a programming language, an interactive operating system, and all the debugging tools to develop applications on LaunchPad Kit, for LaunchPad Kit. The complete source code of 430eForth.asm is only 44 Kbytes long. It is an organic system, which can grow to accommodate any application that MSP430G2553 microcontroller can host. It allows you to read all its CPU and I/O registers, and all its data and program memories. It also allows you to change the I/O registers and memories, and to add new commands to the flash memory. By adding new commands, you can extend the 430eForth system and build a new system which will do what you want it to do.

In 430eForth, I try to reduce the FORTH language to its bare minimum, so that you can learn this programming language quickly, and to use it to do useful work. MSP430G2553, like all the newer microcontrollers available now, contains many powerful and complicated I/O devices, and it takes the MSP430 User's Guide 658 pages to explain them. With 430eForth, you can examine all the I/O registers and modify them to make the I/O devices work the way you want them to work. There is no better way to study the MSP430 User's Guide than to read the book along with 430eForth, modifying the I/O registers and observe what the I/O devices do. 430eForth is a worthy companion to the MSP430 User's Guide.

LaunchPad Kit is an excellent platform for FORTH. FORTH allows you to develop substantial applications quickly and produce high quality code. You write commands in small modules which can be tested exhaustively. Fully tested commands can be used to build more powerful commands at higher conceptual levels, until the last command, which becomes the application. This last command can be used to configure a turnkey system, so that it will be executed when the system boots up. You can do all these things with 430eForth on LaunchPad Kit.

FORTH is a programming paradigm very different from conventional programming languages and operating systems. It can be embedded into a small microcontroller, and empowers you to make the best use of the limited resources available in a microcontroller. I hope you will learn this paradigm and enjoy these benefits:

Integrated operating system and programming language on a small chip
Interactive command interpreter
Incremental compilation of new commands
Bottom up coding and debugging
Naturally structured programming
Ready access to memory and I/O registers
Ease in building turnkey applications

In explaining how this system is constructed, every step in the way, I hope to lay to rest these myths, that computers are complicate, programming languages are complicated, and operating systems are complicated. All these things can be very simple, and can be understood by ordinary people and ordinary engineers. If you understand this 430eForth system completely, the understanding can be carried over

to any computer and microcontrollers.

People using computers are trained to be slaves. You are taught to push certain buttons, and your are taught to push certain keys. Then, you get employed to push buttons and keys to work as slaves. Computers, programming languages, and operating systems are made complicated to enslave people.

Computers are not complicated beyond comprehension. Programming languages and operating systems do not have to be complicated. If you get a sharp knife, you can be the master of your destination. 430eForth is a sharp knife. Go use it.

Conclusions

What I give you in 430eForth is that in about 6000 bytes, you have a programming language, an interactive operating system, and all the debugging tools to develop applications on LaunchPad Kit, for LaunchPad Kit. The complete source code of 430eForth.asm is only 44 Kbytes long. It is an organic system, which can grow to accommodate any application that MSP430G2553 microcontroller can host. It allows you to read all its CPU and I/O registers, and all its data and program memories. It also allows you to change the I/O registers and memories, and to add new commands to the flash memory. By adding new commands, you can extend the 430eForth system and build a new system which will do what you want it to do.

In 430eForth, I try to reduce the FORTH language to its bare minimum, so that you can learn this programming language quickly, and to use it to do useful work. MSP430G2553, like all the newer microcontrollers available now, contains many powerful and complicated I/O devices, and it takes the MSP430 User's Guide 658 pages to explain them. With 430eForth, you can examine all the I/O registers and modify them to make the I/O devices work the way you want them to work. There is no better way to study the MSP430 User's Guide than to read the book along with 430eForth, modifying the I/O registers and observe what the I/O devices do. 430eForth is a worthy companion to the MSP430 User's Guide.

LaunchPad Kit is an excellent platform for FORTH. FORTH allows you to develop substantial applications quickly and produce high quality code. You write commands in small modules which can be tested exhaustively. Fully tested commands can be used to build more powerful commands at higher conceptual levels, until the last command, which becomes the application. This last command can be used to configure a turnkey system, so that it will be executed when the system boots up. You can do all these things with 430eForth on LaunchPad Kit.

FORTH is a programming paradigm very different from conventional programming languages and operating systems. It can be embedded into a small microcontroller, and empowers you to make the best use of the limited resources available in a microcontroller. I hope you will learn this paradigm and enjoy these benefits:

Integrated operating system and programming language on a small chip
Interactive command interpreter
Incremental compilation of new commands
Bottom up coding and debugging
Naturally structured programming
Ready access to memory and I/O registers
Ease in building turnkey applications

In explaining how this system is constructed, every step in the way, I hope to lay to rest these myths, that computers are complicate, programming languages are complicated, and operating systems are complicated. All these things can be very simple, and can be understood by ordinary people and ordinary engineers. If you understand this 430eForth system completely, the understanding can be carried over to any computer and microcontrollers.

People using computers are trained to be slaves. You are taught to push certain buttons, and your are taught to push certain keys. Then, you get employed to push buttons and keys to work as slaves. Computers, programming languages, and operating systems are made complicated to enslave people.

Computers are not complicated beyond comprehension. Programming languages and operating systems do not have to be complicated. If you get a sharp knife, you can be the master of your destination. 430eForth is a sharp knife. Go use it.

Appendix 430eForth Commands

Stack Comments:
Stack inputs and outputs are shown in the form: (input1 input2 ... -- output1 output2 ...)
Stack Abbreviations of Number Types

flag	Boolean flag, either 0 or -1
char	ASCII character or a byte
n	16 bit number
addr	16 bit address
d	32 bit number

Stack Manipulation Commands

?DUP	(n -- n n | 0)	Duplicate top of stack if it is not 0.
DUP	(n1 -- n2)	Duplicate top of stack.
DROP	(n --)	Discard top of stack.
SWAP	(n1 n2 -- n2 n1)	Exchange top two stack items.
OVER	(n1 n2 -- n1 n2 n1)	Make copy of second item on stack.
ROT	(n1 n2 n3 -- n2 n3 n1)	Rotate third item to top.
PICK	(n -- n1)	Zero based, duplicate nth item to top. (e.g. 0 PICK is DUP).
>R	(n --)	Move top item to return stack for temporary storage.
R>	(-- n)	Retrieve top item from return stack.
R@	(-- n)	Copy top of return stack onto stack.
2DUP	(d -- d d)	Duplicate double number on top of stack.
2DROP	(d1 d2 --)	Discard two double numbers on top of stack
DEPTH	(-- n)	Count number of items on stack.

Arithmetic Commands

+	(n1 n2 -- n3)	Add n1 and n2.
-	(n1 n2 -- n3)	Subtract n2 from n1 (n1-n2=n3).
*	(n1 n2 -- n3)	Multiply. n3=n1*n2
/	(n1 n2 -- n3)	Division, signed (n3= n1/n2).
1+	(n -- n+1)	Increment n.
1-	(n -- n-1)	Decrement n.
2+	(n -- n+2)	Add two to n.
2-	(n -- n-2)	Subtract two from n.
2*	(n -- n*2)	Logic left shift.
2/	(n -- n/2)	Logic right shift.
UM+	(n1 n2 -- nd)	Unsigned addition, double precision result.
UM*	(n1 n2 -- nd)	Unsigned multiply, double precision result.
M*	(n n -- d)	Signed multiply. Return double product.
UM/MOD	(nd n1 -- mod quot)	Unsigned division with double precision dividend.
M/MOD	(d n -- mod quot)	Signed floored divide of double by single. Return mod and quotient.
MOD	(n1 n2 -- mod)	Modulus, signed (remainder of n1/n2).
/MOD	(n1 n2 -- mod quot)	Division with both remainder and quotient.
*/MOD	(n1 n2 n3 -- n4 n5)	Multiply and then divide (n1*n2/n3)
*/	(n1 n2 n3 -- n4)	Like */MOD, but with quotient only.
ABS	(n1 -- n2)	If n1 is negative, n2 is its two's complement.
NEGATE	(n1 -- n2)	Two's complement.
MAX	(n1 n2 -- n3)	n3 is the larger of n1 and n2.
MIN	(n1 n2 -- n3)	n3 is the smaller of n1 and n2.
WITHIN	(n1 n2 n3 -- flag)	Return true if n1 is within range of n2 and n3. (n2 <= n1 < n3)
DNEGATE	(d1 -- d2)	Negate double number. Two's complement.
D+	(d1 d2 -- d3)	Add double numbers.
D-	(d1 d2 -- d3)	Subtract double numbers.

D- (d1 d2 -- d3) Subtract double numbers.

Logic and Comparison Commands

AND	(n1 n2 -- n3)	Logical bit-wise AND.
OR	(n1 n2 -- n3)	Logical bit-wise OR.
XOR	(n1 n2 -- n3)	Logical bit-wise exclusive OR.
INVERT	(n1 -- n2)	Bit-wise one's complement.
0<	(n -- flag)	True if n is negative.
U<	(n1 n2 -- flag)	True if n1 less than n2. Unsigned compare.
<	(n1 n2 -- flag)	True if n1 less than n2.
=	(n1 n2 -- flag)	True if n1 equals n2.
>	(n1 n2 -- flag)	True if n1 greater than n2.
D>	(d1 d2 -- flag)	True if d1 greater than d2.

RAM Memory Commands

@	(addr -- n)	Replace addr by number at addr.
C@	(addr -- char)	Fetch least-significant byte only.
!	(n addr --)	Store n at addr.
C!	(char addr --)	Store least-significant byte only.
+!	(n addr --)	Add n to number at addr.
COUNT	(addr1 -- addr+1 char)	Move string count from memory onto stack.
ALLOT	(n --)	Add n bytes to the RAM pointer DP.
HERE	(-- addr)	Address of next available RAM memory location.
PAD	(-- addr)	Address of a scratch area of at least 64 bytes.
TIB	(-- addr)	Address of terminal input buffer.
CMOVE	(addr1 addr2 n --)	Move n bytes starting at memory addr1 to addr2.
FILL	(addr n char --)	Fill n bytes of memory at addr with char.

Flash Memory Commands

I@	(addr -- n)	Replace addr by number at flash memory addr.
IC@	(addr -- char)	Fetch a byte from flash memory addr.
I!	(n addr --)	Store n at flash memory addr.
ICOUNT	(addr1 -- addr+1 char)	Move string count from flash memory onto stack.
IALLOT	(n --)	Add n bytes to the flash memory pointer CP.
ITYPE	(addr n --)	Display a string of n characters in flash starting at address addr.
READ	(addr1 addr2 --)	Read 128 bytes from flash memory addr1 to RAM memory addr2.
WRITE	(addr1 addr2 --)	Write 128 bytes from RAM memory addr1 to flash memory addr2.
ERASE	(addr --)	Erase an 128 byte page in flash memory at addr.
FLUSH	(--)	Write modified flash buffers back to flash memory.

System Variables

'BOOT	(-- addr)	Contain address of application command to boot.
BASE	(-- addr)	Contain radix for number conversion
TMP	(-- addr)	Temporary scratch pad
SPAN	(-- addr)	Contain actual number of characters received by EXPECT
>IN	(-- addr)	Contain character offset into the input stream buffer.
#TIB	(-- addr)	Contain current length of terminal input buffer (TIB).
'TIB	(-- addr)	Contain current address of terminal input buffer (TIB)
'EVAL	(-- addr)	Contain interpreter or compiler to evaluate a command.
HLD	(-- addr)	Contain pointer to numeric string under construction.
CONTEXT	(-- addr)	Contain name field address of last command in dictionary
CP	(-- addr)	Contain first free address in flash memory
DP	(-- addr)	Contain first free address in RAM memory
LAST	(-- addr)	Contain name field address of command under compilation

Terminal Input-Output Commands

EMIT	(char --)	Display char.
KEY	(-- char)	Get an ASCII character from the keyboard.
?KEY	(-- char -1 | 0)	Return an ASCII character from the keyboard and a true flag. Return false flag if no character available.
.	(n --)	Display number n with a trailing blank.
U.	(n --)	Display an unsigned integer with a trailing blank.
.R	(n1 n2 --)	Display signed number n1 right justified in n2 character field.
U.R	(n1 n2 --)	Display unsigned number n1 right justified in n2 character field.
?	(addr --)	Display contents at memory addr.
<#	(--)	Start numeric output string conversion.
#	(n1 -- n2)	Convert next digit of number and add to output string
#S	(n --)	Convert all significant digits in n to output string.
HOLD	(char --)	Add char to output string.
SIGN	(n --)	If n is negative, add a minus sign to the output string.
#>	(xd -- addr n)	Terminate numeric string, leaving addr and count for TYPE.
CR	(--)	Display a new line.
SPACE	(--)	Display a space.
SPACES	(n --)	Display n spaces.
EXPECT	(addr n --)	Accept n characters into buffer at addr.
CHAR	(-- char)	Parse next command and return its first character.
TYPE	(addr n --)	Display a string of n characters starting at address addr.
BL	(-- 32)	Return ASCII Blank character.
DECIMAL	(--)	Set number base to decimal.
HEX	(--)	Set number base to hexadecimal.

Compiler and Interpreter Commands

:<name>	(--)	Begin a colon definition of <name>.
;	(--)	Terminate execution of a colon definition.
CREATE <name>	(--)	Dictionary entry with no parameter field space reserved.
VARIABLE <name>	(--)	Defines a variable. At run-time, <name> leaves its address.
CONSTANT <name>	(n --)	Defines a constant. At run-time, n is left on the stack.
,	(n --)	Compile n to the dictionary in flash memory
IMMEDIATE	(--)	Cause last-defined command to execute even within a colon definition.
COMPILE <name>	(--)	<name> is compiled to dictionary.
[COMPILE] <name>	(--)	Immediate command <name> is compiled to dictionary.
LITERAL	(n --)	Compile literal number n. At run-time, n is pushed on the stack.
[	(--)	Switch from compilation to interpretation.
]	(--)	Switch from interpretation to compilation.
WORD<text>	(char -- addr)	Get the char delimited string <text> from the input stream and leave as a counted string at addr.
(comment)	(--)	Ignore comment text.
\ comment	(--)	Ignore comment till end of line.
." <text>"	(--)	Compile <text> message. At run-time display text message.
.(<text>)	(--)	Display <text> from the input stream.
$" <text>"	(-- addr)	Compile <text> message. At run-time return its address.
ABORT" <text>"	(flag --)	Compile <test> message. At run-time display message and abort if flag is true. Otherwise, ignore message and continue.
COLD	(--)	Start eForth system.
QUIT	(--)	Return to interpret mode, clear data and return stacks.
QUERY	(--)	Accept input stream to terminal input buffer.
NAME>	(addr1 -- addr2)	Traverse name field at addr1 and return code field address

		addr2.
NUMBER?	(addr -- n -1 | addr 0)	Convert a number string to integer. Push a flag on tos.
EXECUTE	(addr --)	Execute command definition at addr.
@EXECUTE	(addr --)	Execute command definition whose execution address is in addr.
EXIT	(--)	Terminate execution of a colon definition.

Compiler Structure Commands

IF	(flag --)	If flag is zero, branches forward to ELSE or THEN.
ELSE	(--)	Branch forward to THEN.
THEN	(--)	Terminate a IF-ELSE-THEN structure.
FOR	(n --)	Setup loop with n as index. Repeat loop n+1 times.
NEXT	(--)	Decrement loop index by 1 and branch back to FOR. Terminate FOR-NEXT loop when index is negative.
AFT	(--)	Branch forward to THEN in a loop to skip the first round
BEGIN	(--)	Start an indefinite loop.
AGAIN	(--)	Branch backward to BEGIN.
UNTIL	(flag --)	Branch backward to BEGIN if flag is false. If flag is true, terminate BEGIN-UNTIL loop.
WHILE	(flag --)	If flag is false, branch forward to terminate BEGIN-WHILE-REPEAT loop. If flag is true, continue execution till REPEAT.
REPEAT	(--)	Resolve WHILE clause. Branch backward to BEGIN.

Utility Commands

' <name>	(-- addr)	Look up <name> in the dictionary. Return execution address.
WORDS	(--)	Display all eForth commands
DUMP	(addr --)	Dump 128 bytes of RAM memory starting from addr.
IDUMP	(addr --)	Dump 128 bytes of flash memory starting from addr.
.S	(--)	Dump the parameter stack.