

Appnote005

The Morse Gadget

or
How to Program with AmrFORTH

INTRODUCTION

In this application note we follow the development path of a relatively straightforward project in Forth which takes serial data from any source and converts this text into international Morse Code with programmable transmission speed and tone.

REQUIREMENTS

It is assumed that you have a working AM Research Gadget, a Graphical User Interface operating system such as MicroSoft Windows, Linux, Apple OSX, or FreeBSD. Install the Gadget according to directions contained on the CD or downloaded from our website. An excellent reference for those who wish to build their own hardware is at:

<http://www.amresearch.com/software/v6/index.html>

Our examples are written in Linux using the KDE desktop but are almost identical to other Windowing environments except for paths.

START HERE

Once you have installed amrFORTH you will want to use it to complete some project. For example let's design a Morse Code Translator using a gadget300. This gadget300 will receive character input via its RS232 port, and output morse code to a piezo transducer. How do you get started?

The first thing we need to do is to start the amrFORTH GUI (Graphical User Interface). From there we can create a project directory. There are at least two ways to get amrFORTH started. From either Windows or Linux/BSD you can use a command line, either DOS or XTERM. From the command line you:

```
"cd \amrforth\v6\300\example"      for Windows, or  
"cd ~/amrforth/v6/300/example"    for Linux/BSD.
```

From that directory you type "amrf <enter>", in either operating system, to start the GUI. Alternatively you can use a graphical file manager, "My Computer" or "Explorer" in Windows, or "Konqueror" in Linux/BSD. Just double click on the icon for "amrf" or "amrf.bat" in the directory mentioned above to execute the script.

Once the application is open pull down the "File" menu and choose "New Project". You will get a dialog that helps you select a directory path for your

new project.

For this example we will make:

"\Projects\morse"	for Windows, and
"~/Projects/morse"	for Linux/BSD.

Hint, using KDE with Linux, the command line to enter in the "execute" field should be this:

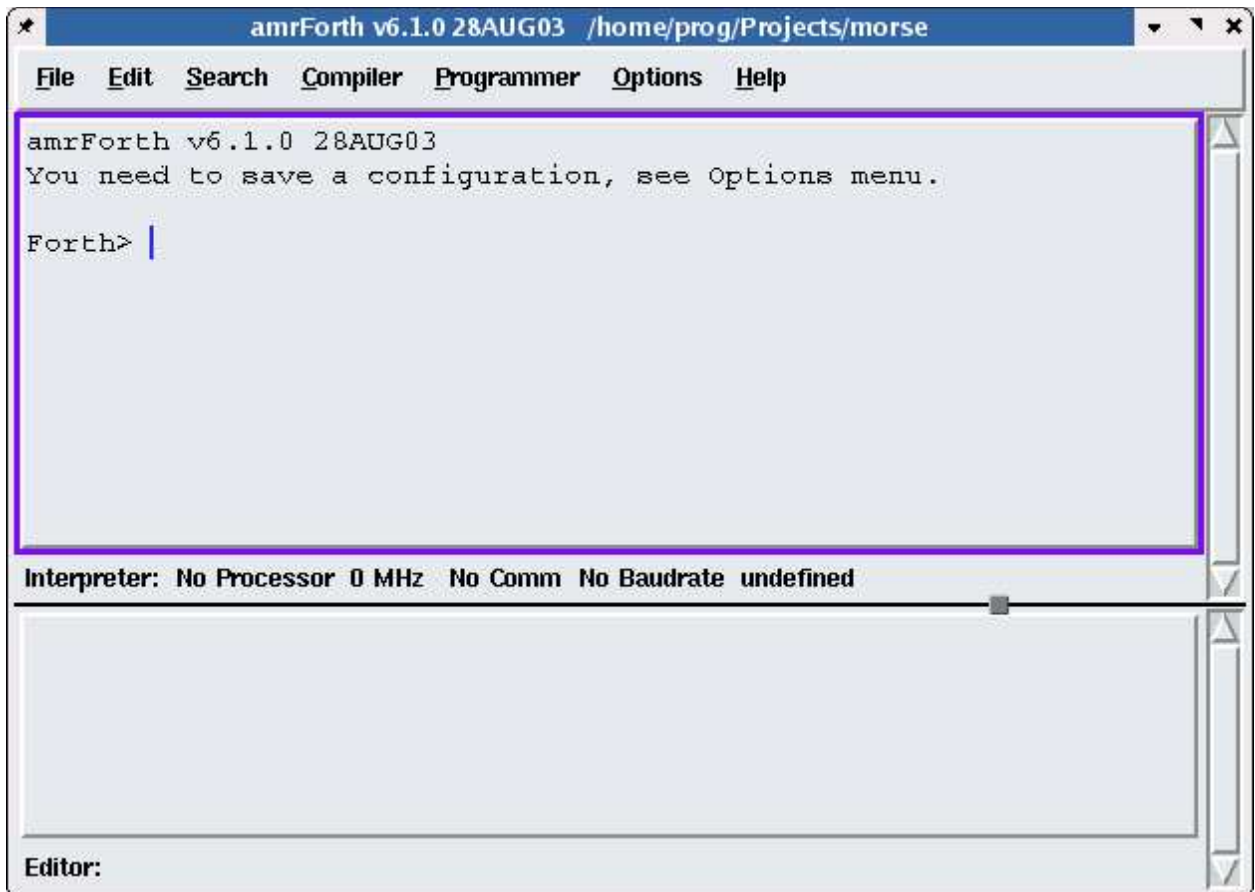
```
"cd ~/Projects/morse; amrf"
```

You need to be in the project directory when you execute the "amrf" script, otherwise the wrong config files will be found and used. Likewise in Windows it is important to enter:

```
"\Projects\morse"
```

in the "working" directory field of the icon shortcut properties.

At this point, the amrFORTH GUI looks like this depending upon your screen resolution and settings:



Note that the project directory path is in the GUI title bar.

Next we need to configure the project directory appropriately for this project. We will use a Gadget300. Before jumping in let's think about this for a moment, if the data input is running at computer speed but output is limited to a programmable Morse speed we could might receive the entire 1,000 pages of War and Peace before the first paragraph was sounded! We must protect against missing characters and obviously we can't expect a \$30 Gadget to buffer massive amounts of data.

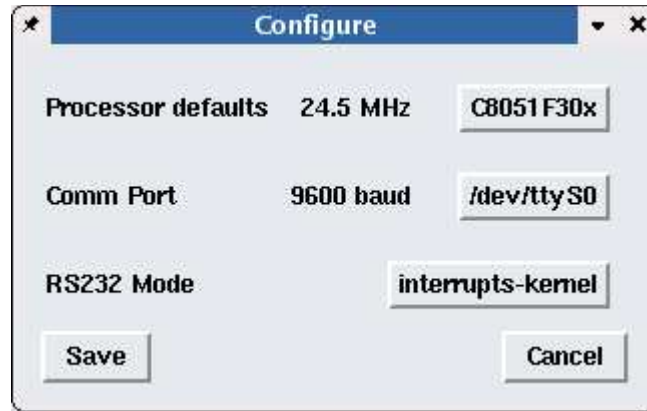
A standard option to handle situations like this in serial communications is called "XON/XOFF." In XON/XOFF software handshaking is implemented so the data source (or master) asks for permission before sending information. If the recipient, a Gadget300 in this case (aka slave), is full then it waits before sending a ready to the host. The advantage of xon/xoff to hardware handshaking is that only three wires, one transmitter, one receiver and one ground are required, it is economical in hardware. The alternative, hardware handshaking, requires 4 additional digital signals, wires, 2 more RS232 level shift transmitters and 2 more RS232 receivers plus associated s/w to support them. In this case we'll go with XON/XOFF.

All serial communication programs, Hyperterm, Telix, Minicom, etc., have an options menu which permits the user to set xon/xoff active. When using the Morse Gadget set this parameter active. While you're here set protocols and speed to 9600, 8N1 for convenience and save that configuration for posterity.

On the Gadget side, let's configure with serial interrupts in order not to miss any characters in the serial input stream in case the Gadget is busy beeping out a bunch of characters and choose the correct comm port, in our case /dev/ttyS0 since we are using Linux, or com1 in Windows. The "Options" menu looks like this:



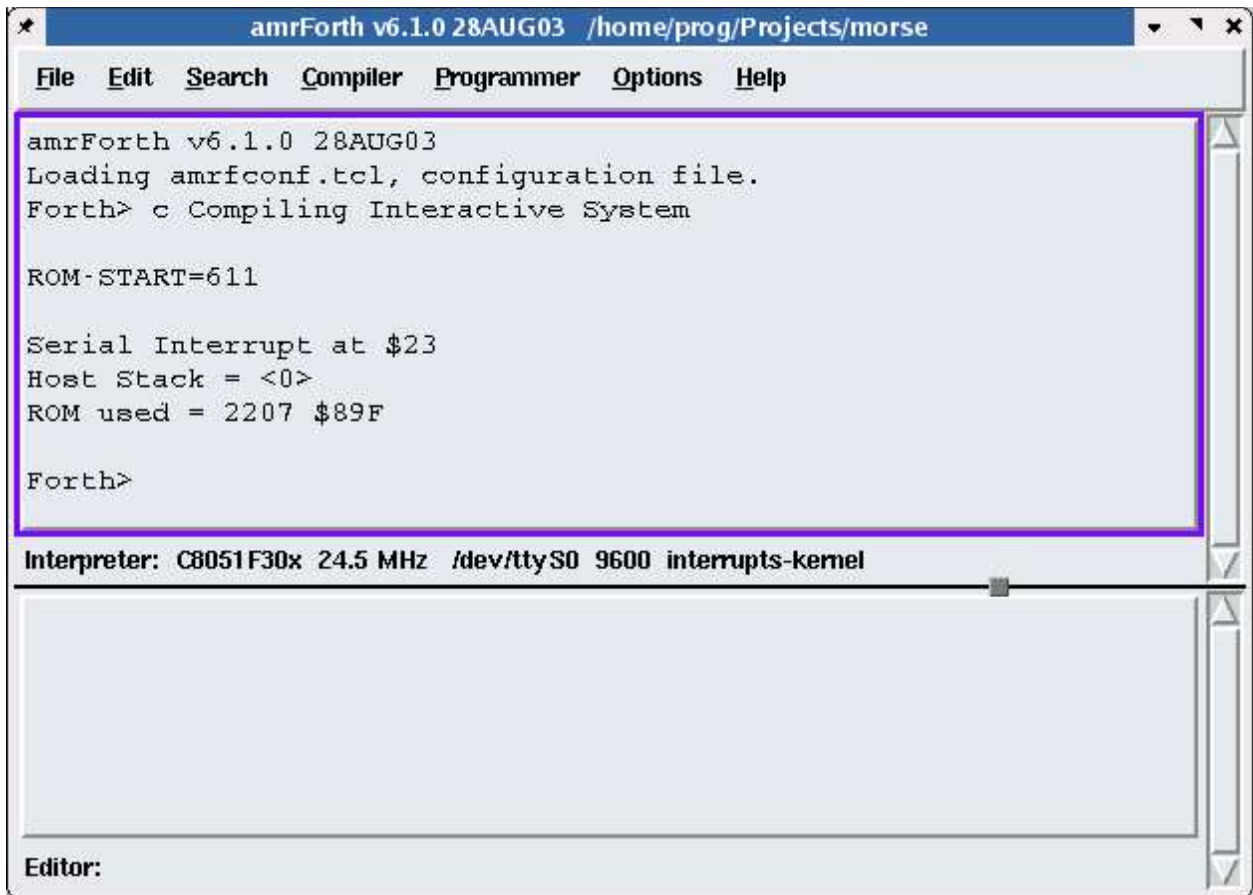
Next choose the "Configure" choice in our menu line to get this dialog box:



(or type “config” on the amrFORTH GUI command line)

For this project we want the C8051F30x chip, com1 or /dev/ttyS0 or com2 if that works better for you, and the interrupts-kernel. Click the “Save” button to preserve this configuration in the current project directory.

Now it's time for some hardware. Get your gadget300 and plug it into the middle slot of your Gadget Motherboard, J2, plug the modular serial connector into the COM1 (or ttyS0 in Linux) of your PC and into the serial port labeled J6 on the Gadget motherboard. Apply power with the included wall wart power supply. The configuration process above has already installed a few files in your project directory, everything you need to compile and download. From the GUI window command line type “c” (without the quotes) for compile then press enter. You should see this on your screen:



Immediately after your "c" is a message from the compiler telling you this will be an interactive system rather than a turnkey system. This means you will be able to test individual forth words on the target after downloading. The compiler reports the address where code starts in ROM, after the interrupt vector table because the address is different for different Gadget capabilities. Next is the serial interrupt address report (remember that we configured the kernel with a serial interrupt).

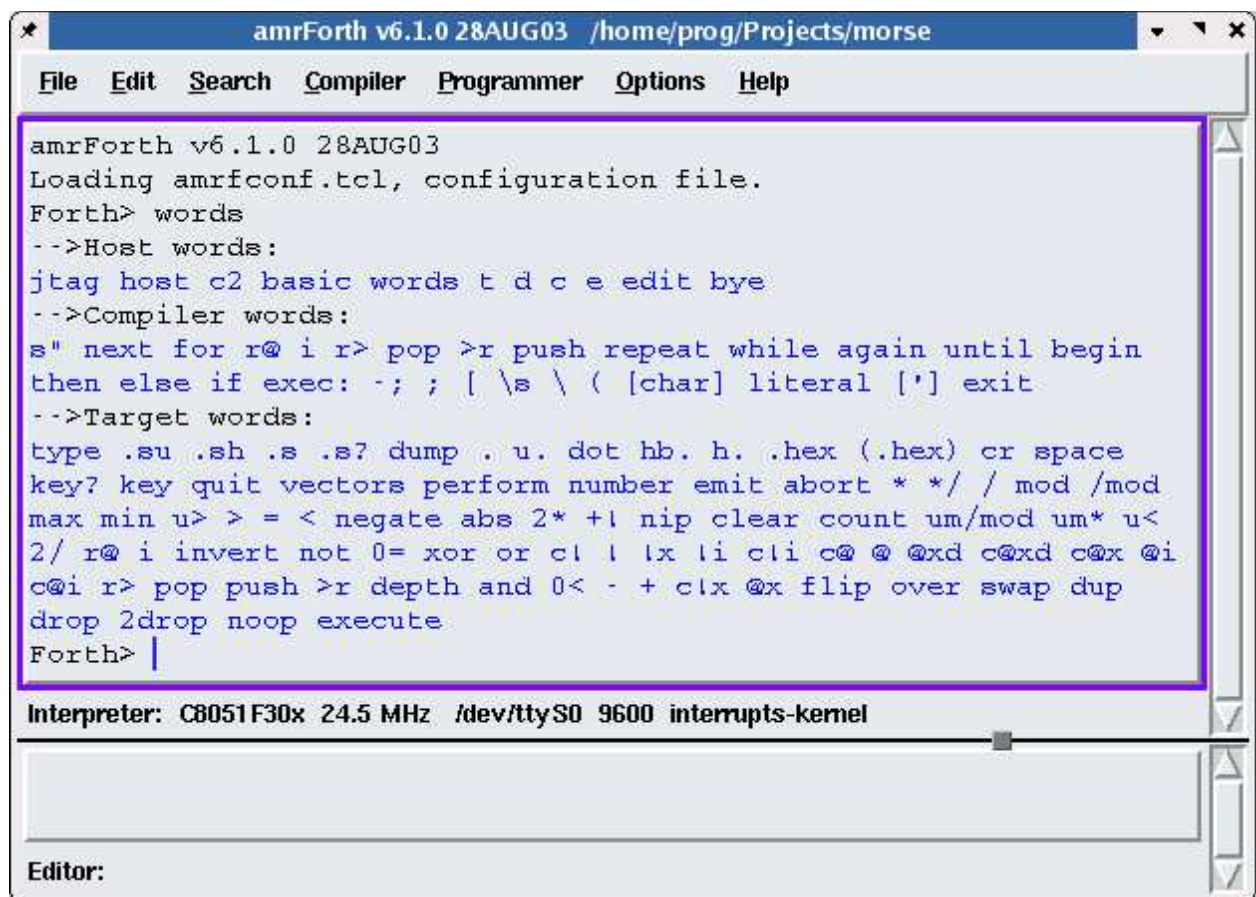
After compilation the host stack is reported as empty. This is good. If the PC host has anything on its stack after compiling you know something is wrong and need to determine where the stack item came from, somewhere in your source code. We see that 2275 bytes of ROM have been used, that's program memory. The same number is repeated in hexadecimal for those who prefer it.

You might be wondering what has been compiled, since we haven't created any source code yet. When you created this project directory, the file "job.fs" was also created and placed in this directory. "job.fs" is a load file for amrFORTH. It is always loaded when you compile with "c" (Compile) or "t" (Turnkey). You could also execute "c" and "t" by mousing the "Compiler" menu and choosing "c" or "t". For a very small project let's place all source code in "job.fs".

Usually you will want to divide your project up into several files by function, then load them from “job.fs” with the line:

```
include morse.fs
```

In this case when you ran the compiler the “job.fs” file was empty, so only the amrFORTH kernel was compiled. The kernel is the smallest set of forth words you can compile by default. These are the words you will be using to build your application. Typing “words” at the command line prints a list of all words currently known by the Gadget. This is the same as saying that it is also a list of all “commands” available to the Gadget.



The screenshot shows a window titled "amrForth v6.1.0 28AUG03 /home/prog/Projects/morse". The window has a menu bar with "File", "Edit", "Search", "Compiler", "Programmer", "Options", and "Help". The main text area displays the following output:

```
amrForth v6.1.0 28AUG03
Loading amrfconf.tcl, configuration file.
Forth> words
-->Host words:
jtag host c2 basic words t d c e edit bye
-->Compiler words:
s" next for r@ i r> pop >r push repeat while again until begin
then else if exec: -; ; [ \s \ ( [char] literal ['] exit
-->Target words:
type .su .sh .s .s? dump . u. dot hb. h. .hex (.hex) cr space
key? key quit vectors perform number emit abort * */ / mod /mod
max min u> > = < negate abs 2* +! nip clear count um/mod um* u<
2/ r@ i invert not 0= xor or cl l lx li cli c@ @ @xd c@xd c@x @i
c@i r> pop push >r depth and 0< - + clx @x flip over swap dup
drop 2drop noop execute
Forth> |
```

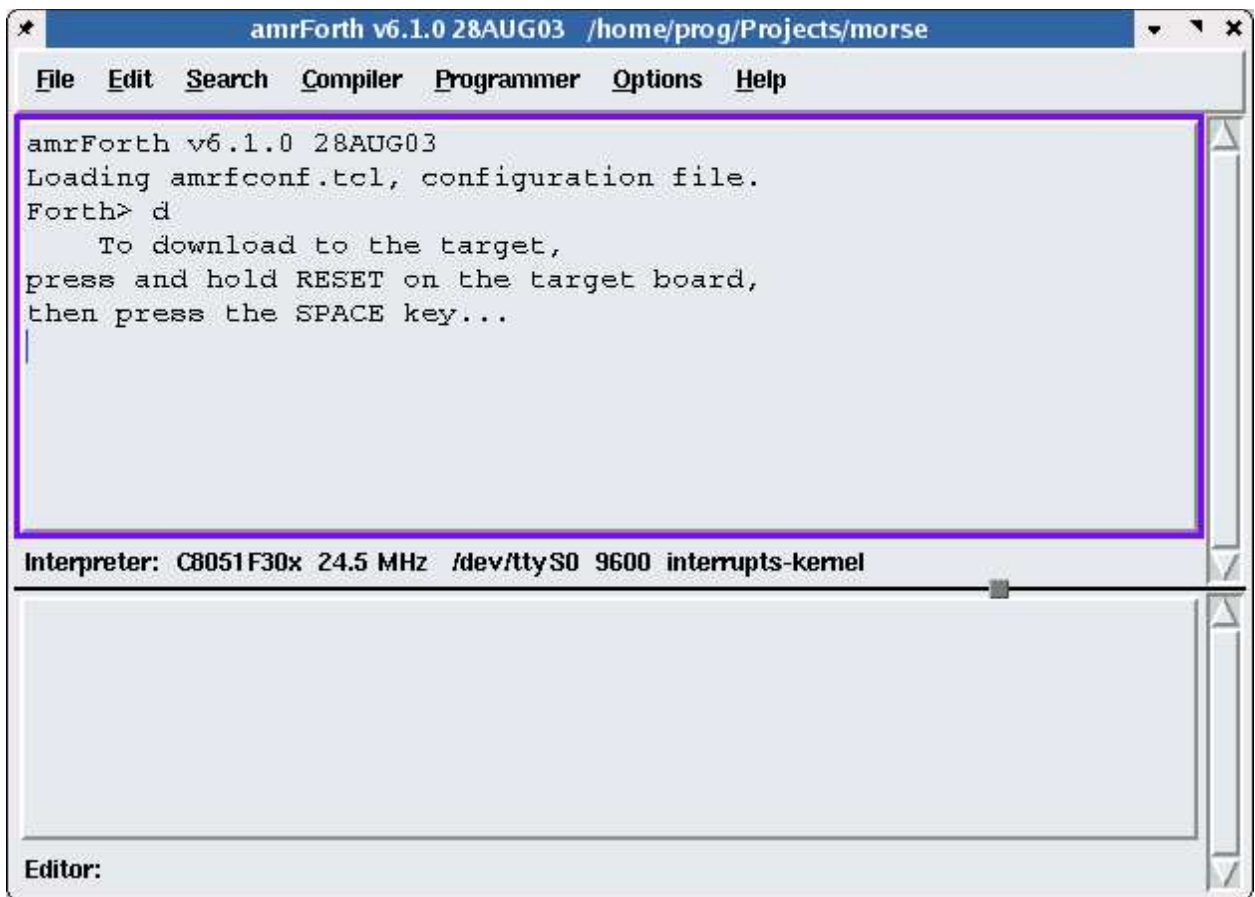
Below the text area, there is a status bar that reads "Interpreter: C8051F30x 24.5 MHz /dev/ttyS0 9600 interrupts-kernel". At the bottom of the window, there is an "Editor:" label followed by a large empty text area.

The “Target words” list is essentially a list of assembler subroutines known to the compiler which can be run directly from the command line. Words in the “Compiler words” list are known only to the compiler at compile time. These words cannot be run interactively, but you will be using them in your program source.

Note the list of “Host words:”, namely jtag, host, c2, basic, words, t, d, c, e, edit, and bye. These are words that execute only on the PC host, not on the gadget.

The amrFORTH GUI has five modes or vocabularies, named host, forth, basic, jtag, and c2. So far we have only seen the forth> prompt, because we began in that mode. By typing “host” and then “words” at the command line you can see what is available in the host mode. This is meant for getting help, configuring, and moving to other modes. The forth mode is meant for interactive testing of forth programs on the Gadget. The basic mode is for testing basic programs interactively on the Gadget. Jtag and c2 modes are for using the jtag and c2 interfaces to program a new bootloader. We will only need to deal with the forth mode for this project.

The bare kernel has been compiled. Let' s download it to the gadget and see how the interpreter works. From the amrFORTH command line type “d” and return. The screen should look like this:



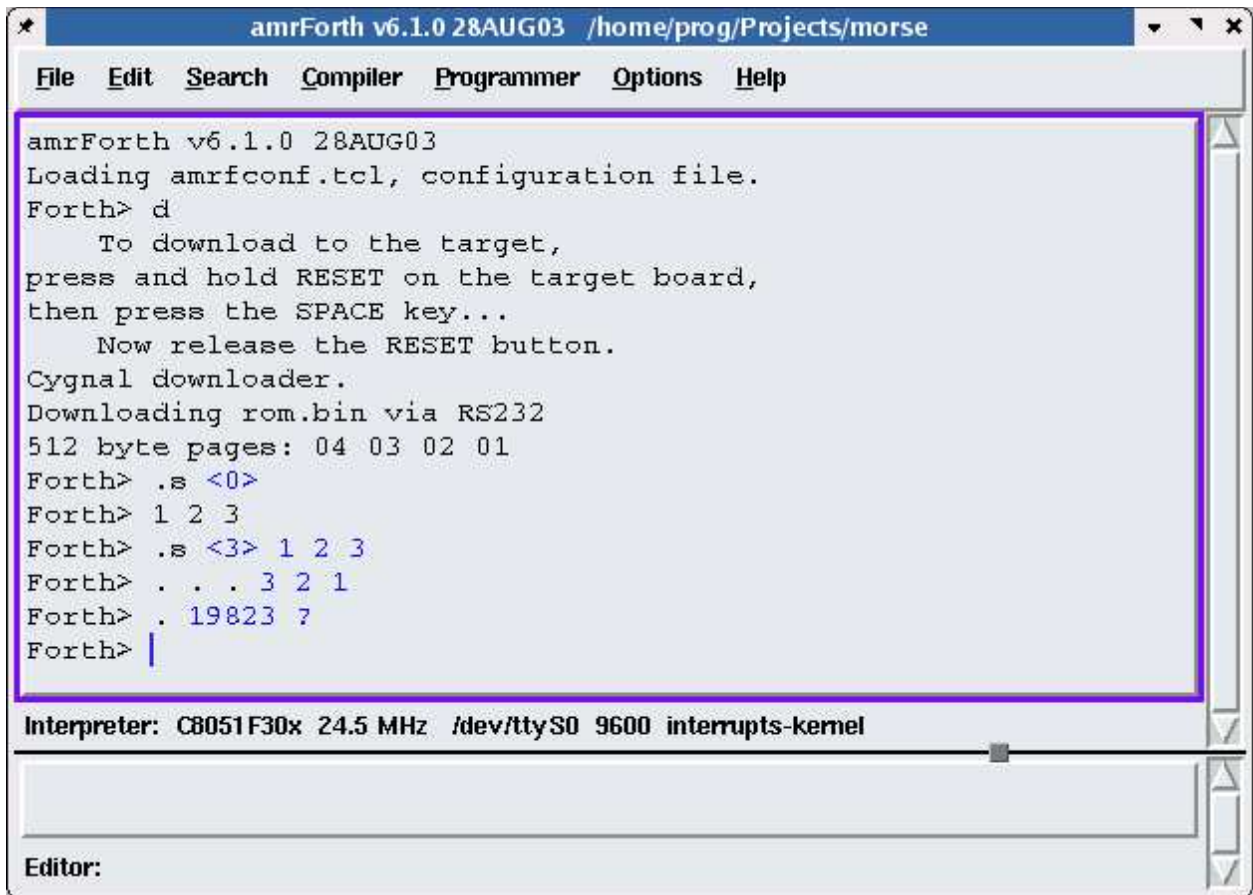
Be sure your gadget is plugged in and the power LED is lit. Press the reset button labeled SW1 on the board, holding it until you have hit the space bar on your host computer. The host and the gadget will do some handshaking and the object code for your program, in this case just the bare kernel, will be loaded into program memory on the gadget. The GUI command line will look like this now:

The screenshot shows a terminal window titled "amrForth v6.1.0 28AUG03 /home/prog/Projects/morse". The window has a menu bar with "File", "Edit", "Search", "Compiler", "Programmer", "Options", and "Help". The main text area contains the following output:

```
amrForth v6.1.0 28AUG03
Loading amrfconf.tcl, configuration file.
Forth> d
    To download to the target,
press and hold RESET on the target board,
then press the SPACE key...
    Now release the RESET button.
Cygna1 downloader.
Downloading rom.bin via RS232
512 byte pages: 04 03 02 01
Forth>
```

Below the main text area, there is a status bar that reads "Interpreter: C8051F30x 24.5 MHz /dev/ttyS0 9600 interrupts-kernel". At the bottom of the window, there is an "Editor:" label.

Which indicates that four 512 byte pages have been downloaded to the gadget. Since we are already in forth mode, we can immediately test the downloaded code directly on the gadget. Type "words" to be reminded of the available forth words. Remember though, that many of the words expect parameters on the data stack when they are executed. For example, if you type "." without putting anything on the data stack, the gadget will print the number that lies under the bottom of the stack, followed by a question mark to indicate a stack underflow. To test the interpreter, type "1 2 3 ." then try printing the numbers with ".". Here' s what you should see:



The screenshot shows a window titled "amrForth v6.1.0 28AUG03 /home/prog/Projects/morse". The menu bar includes File, Edit, Search, Compiler, Programmer, Options, and Help. The main text area contains the following text:

```
amrForth v6.1.0 28AUG03
Loading amrfconf.tcl, configuration file.
Forth> d
    To download to the target,
press and hold RESET on the target board,
then press the SPACE key...
    Now release the RESET button.
Cygna! downloader.
Downloading rom.bin via RS232
512 byte pages: 04 03 02 01
Forth> .s <0>
Forth> 1 2 3
Forth> .s <3> 1 2 3
Forth> . . . 3 2 1
Forth> . 19823 7
Forth> |
```

Below the main text area, a status bar reads: "Interpreter: C8051F30x 24.5 MHz /dev/ttyS0 9600 interrupts-kernel". At the bottom, there is an "Editor:" label and an empty text area.

We know now that we have a working system, including compiler, downloader, and interpreter. Let' s get started with that morse code project.

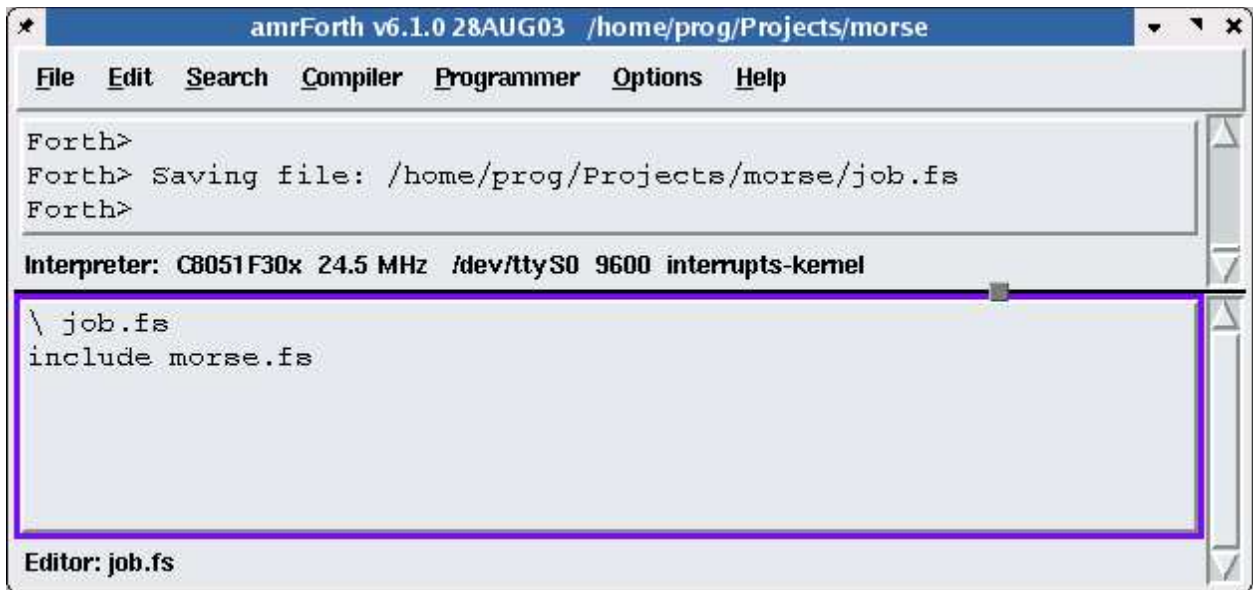
We like to start a new project from the top down. Every project will have startup code followed by an endless loop. Your program will most likely not have an exit point, since there is no operating system to return to. You gadget will be running this program and only this program.

Move to the editor window either by using the mouse or by pressing <shift>right or <shift>left arrow. Here' s a template that should begin any gadget program.

```
amrForth v6.1.0 28AUG03 /home/prog/Projects/morse
File Edit Search Compiler Programmer Options Help
Forth>
Interpreter: C8051F30x 24.5 MHz /dev/ttyS0 9600 interrupts-kernel
\ morse.fs  Morse Code Generator
: init ( - ) ;
: go ( - )
  init
  begin
  again -;|
Editor:
```

You always need to initialize something. We call that “init” for short. The main program is always called “go” because the amrFORTH compiler installs the word called “go” as the program’ s entry point, after the cold start that is. The very curious should look at the the file “end8051.fs” to see how this is done. The main program will settle into an endless loop after initializing, that’ s the begin/again loop. Note the word ends with “-;” instead of just “;”. “-;” ends a definition without compiling a return or “exit”. Since the loop is endless there will never be an exit, and this saves some program memory.

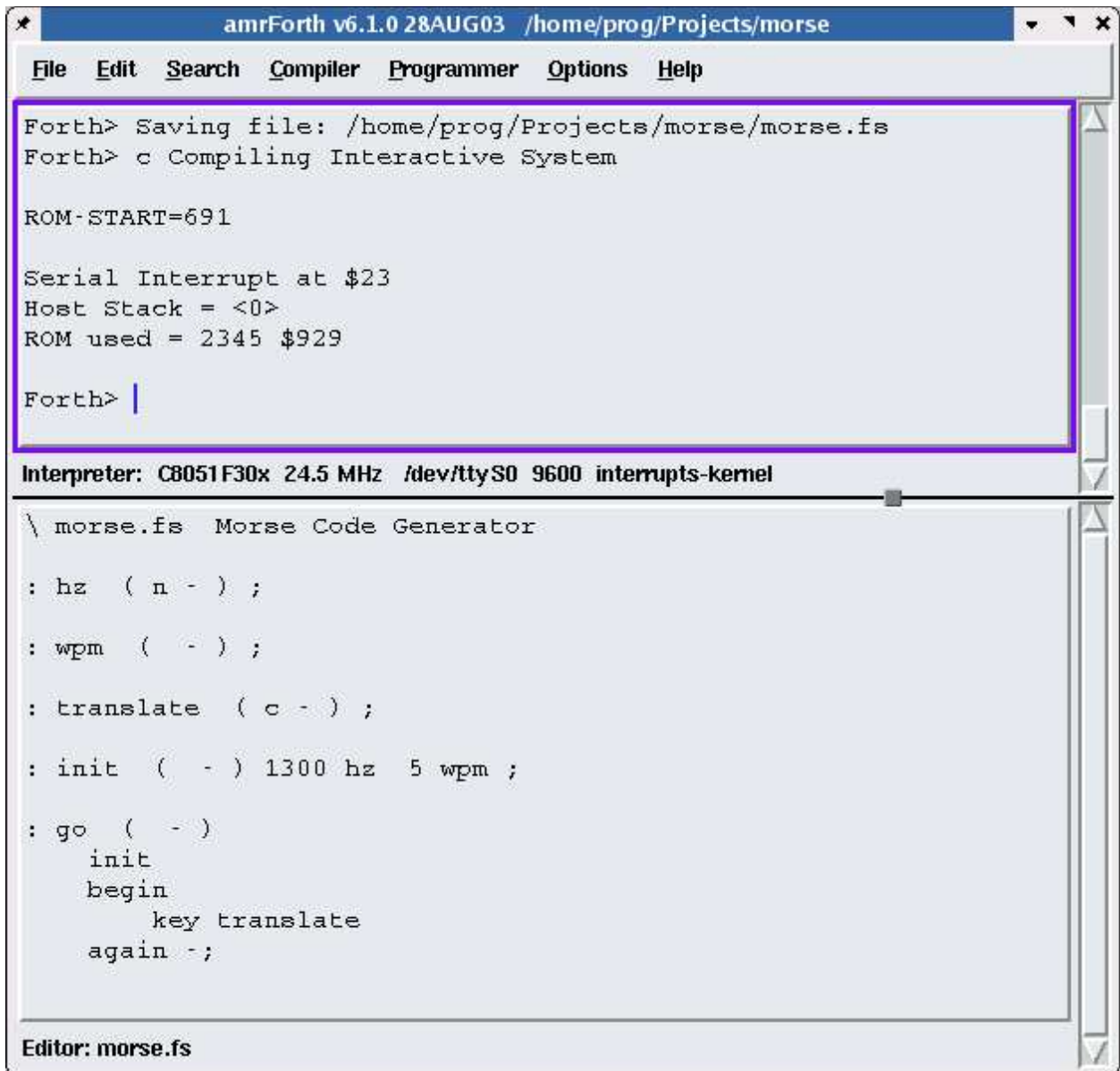
Pull down the “File” menu and choose “Save As”. Fill in “morse.fs” in the resulting dialog. Now you have a source file named “morse.fs” in your project directory. Pull down “File” then “Open” and choose the file “job.fs”. Make job.fs look like this:



Save this file by pressing <control>S or by choosing “Save” on the “File” menu. You will see the line at the Forth> prompt reporting that the file has been saved.

Now we write down in a very high level way, you might say in psuedo-code, what the program will need to do. It will need to get a character over the serial port. AmrFORTH already has a word to do that, called “key”. Once the key has been received the program will translate that key into the dits and dahs of morse code. We’ ll invent the word translate and for the moment we don’ t worry about how it works. We only know that is takes one character as input on the data stack, and returns nothing.

We can imagine now that we might want to specify the frequency of the sound and the timing in words per minute. We invent two other new words called “hz” for hertz, and “wpm” for words per minute. Once again we ignore the inner workings of the words and simply specify their stack effects. Notice in the following graphic that this is not just psuedo-code. Though the program as written does nothing useful yet, it can be compiled as is without error.



```
amrForth v6.1.0 28AUG03 /home/prog/Projects/morse
File Edit Search Compiler Programmer Options Help

Forth> Saving file: /home/prog/Projects/morse/morse.fs
Forth> c Compiling Interactive System

ROM-START=691

Serial Interrupt at $23
Host Stack = <0>
ROM used = 2345 $929

Forth> |

Interpreter: C8051F30x 24.5 MHz /dev/ttyS0 9600 interrupts-kernel

\ morse.fs Morse Code Generator

: hz ( n - ) ;

: wpm ( - ) ;

: translate ( c - ) ;

: init ( - ) 1300 hz 5 wpm ;

: go ( - )
  init
  begin
    key translate
  again -;

Editor: morse.fs
```

The word “translate” can be very simple. The character on the stack should be used as an index into a table or case statement to decide what to do. AmrFORTH has a word called “exec:” which can be thought of as a computed goto. The number on the stack before “exec:” is used to index into the list of words or subroutines after “exec:” up until a “;” or “-;” is encountered. Exec: itself does no bounds checking, so the programmer should do that explicitly. Exec: is not standard forth, but is derived from a very popular free forth for DOS called FPC.

Further, let's assume we can program words to make the short and long sounds of dit and dah, and call them “.” and “-”, or dot and dash. Now we can define words for each of the letters in terms of dots and dashes, and put those words into the translation table.

```
amrForth v6.1.0 28AUG03 /home/prog/Projects/morse
File Edit Search Compiler Programmer Options Help

ROM-START=691

Serial Interrupt at $23
Host Stack = <0>
ROM used = 2611 $A33
redefined . redefined -

Forth> Saving file: /home/prog/Projects/morse/morse.fs
Forth> Saving file: /home/prog/Projects/morse/morse.fs
Forth>

Interpreter: C8051F30x 24.5 MHz /dev/ttyS0 9600 interrupts-kernel

: K - . - ;
: L . - . . ;
: M - - ;
: N - . ;
: O - - - ;
: P . - - . ;
: Q - - . - ;
: R . - . ;
: S . . . ;
: T - ;
: U . . - ;
: V . . . - ;
: W . - - ;
: X - . . - ;
: Y - . - - ;
: Z - - . . ;

: translate ( c - )
  [char] A negate + exec:
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
  - ;

Editor: morse.fs
```

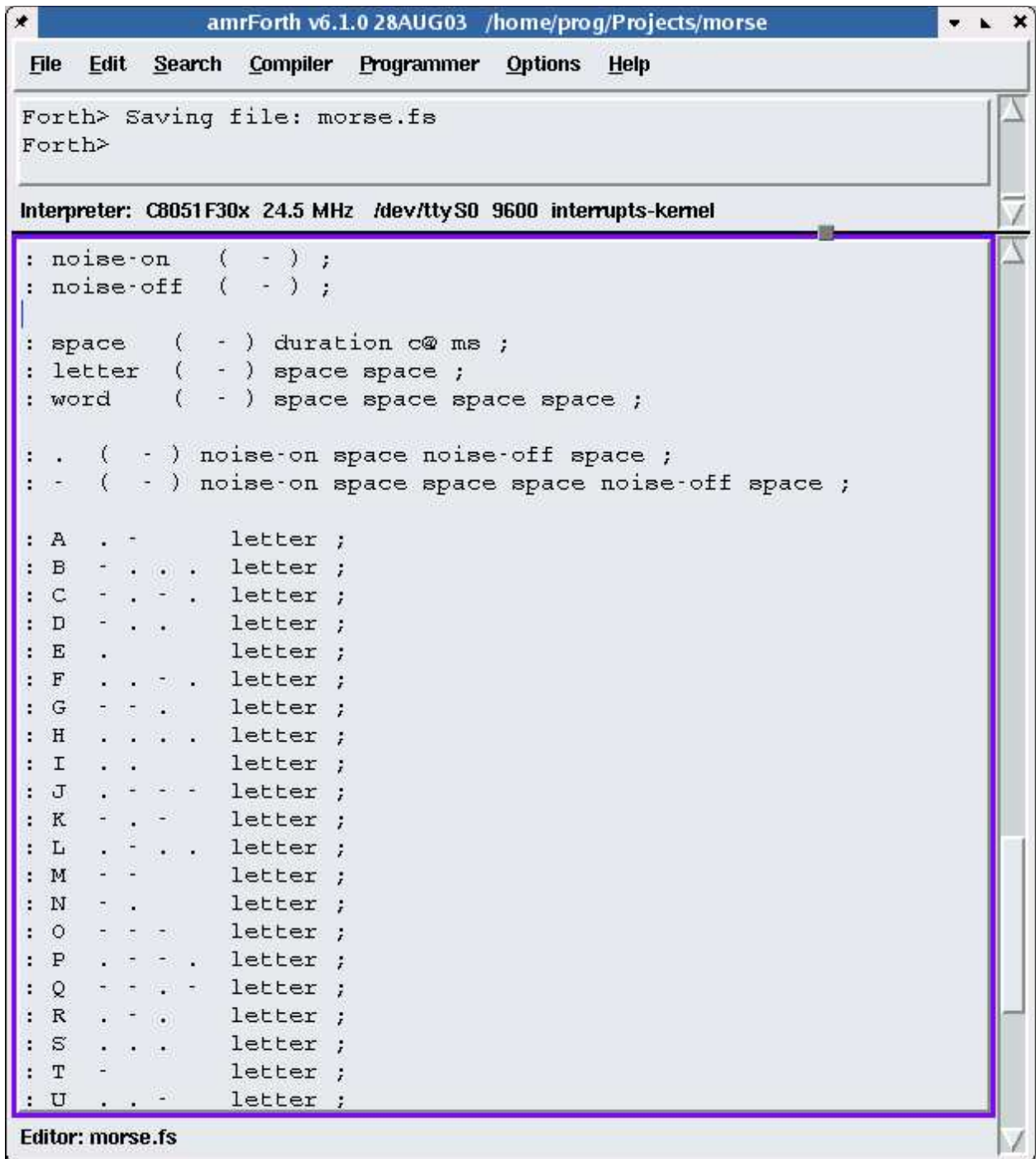
Maybe it bothers you that we have made new words called ' ' . ' ' and ' ' - ' ' when amrFORTH already has those words doing something else. Well you could call the words ' ' dit' ' and ' ' dah' ' for example, but ' ' . ' ' and ' ' - ' ' read so much better. The code for each letter is self documenting. The compiler itself doesn't care. There are no reserved words in forth. We simply decide that we can get along without the original words ' ' . ' ' and ' ' - ' ' for this program and redefine them. That's why we use ' ' negate +' ' in the first line of ' ' translate' ' instead of ' ' - ' ' .

Note that the program still compiles without error, and still actually does nothing.

To be honest though, in the interpreter there *are* some reserved words. The “host” words mentioned above take precedence over the target words in the interpreter. This means that if you compile and download the program above and try to run the word “c” or “t” for example, you will end up recompiling the program rather than testing the words “c” or “t”. As far as the compiler is concerned, there is also a small set of reserved words. Those are the “Compiler words” mentioned above. They will take precedence over any target words with the same names during compilation.

There is a little more top down work we can do here. We know after researching morse code, that there is a fundamental unit of timing which we will call a “space”. There is a single space between each dit and/or dah. The dit takes the time of one space and the dah takes the time of three spaces. Three spaces separate letters and seven spaces separate words. Once “space” is defined, all the other timing can be based on it.

In order to define “space” we will pretend for a moment that we have the word called “ms” which will kill time for a given number of milliseconds. Though not a standard forth word, “ms” is commonly found in forths that are able to keep time. A variable called “duration” can be loaded with the number of milliseconds in each “space”. Here' s how that might work:



```
amrForth v6.1.0 28AUG03 /home/prog/Projects/morse
File Edit Search Compiler Programmer Options Help

Forth> Saving file: morse.fs
Forth>

Interpreter: C8051F30x 24.5 MHz /dev/ttyS0 9600 interrupts-kernel

: noise-on ( - ) ;
: noise-off ( - ) ;

: space ( - ) duration c@ ms ;
: letter ( - ) space space ;
: word ( - ) space space space space ;

: . ( - ) noise-on space noise-off space ;
: - ( - ) noise-on space space space noise-off space ;

: A . - letter ;
: B - . . . letter ;
: C - . - . letter ;
: D - . . letter ;
: E . letter ;
: F . . - . letter ;
: G - - . letter ;
: H . . . . letter ;
: I . . letter ;
: J . - - - letter ;
: K - . - letter ;
: L . - . . letter ;
: M - - letter ;
: N - . letter ;
: O - - - letter ;
: P . - - . letter ;
: Q - - . - letter ;
: R . - . letter ;
: S . . . letter ;
: T - letter ;
: U . . - letter ;

Editor: morse.fs
```

“.” and “-” now end with “space” because we know there will always be at least the one space between dits and dahs. The letter space is equal to three intra-character spaces, but since each dit and dah already uses one space, the word “letter” only needs to use two more to make three. Similarly the word “word” which makes the word space is executed only after the “letter” has used three spaces, so it only needs four more. Note that we also invent the words “noise-on” and “noise-off” to indicate where the gadget is making noise and when it’s

being silent.

We' ve mapped out quite a bit of the structure of our program already, but we' re not going to hear any morse code until we address the gadget hardware. It' s time to switch from “top down” to “bottom up”. We want to hear the dit when we type “.” and the dah when we type “-”.

The Cygnal C8051F300 chip which is used in the gadget300 has a 16 bit timer, timer2, which can be used along with a simple interrupt routine to make an square wave of some given frequency on an output port bit that can be connected to a piezo transducer. We learn in the data sheet found on Cygnal' s website, that timer2 can be set up in 16 bit auto-reload mode, such that it overflows and triggers an interrupt at a frequency determined by the reload registers. Once we know the relationship between actual frequency and the value of the reload registers, we will be able to code the word “hz” which was a stub above. Refer to this screen in the following discussion:

The screenshot shows a window titled "amrForth v6.1.0 28AUG03 /home/prog/Projects/morse". The menu bar includes File, Edit, Search, Compiler, Programmer, Options, and Help. The main text area contains the following code:

```
Forth>

Interpreter: C8051F30x 24.5 MHz /dev/ttyS0 9600 interrupts-kernel

\ morse.fs Morse Code Generator

code enable-sound ( - )
  $04 # P0MDOUT orl \ pin 0.2 is push/pull output.
  next c;

label t2-interrupt
  2 .P0 cpl \ Toggle the piezo transducer pin.
  7 .TMR2CN clr \ Clear the interrupt bit.
  reti c;
t2-interrupt $2b int1 \ Install at interrupt vector.

code steps ( n - )
  Apop A cpl A TMR2RLH mov
  Apop A cpl A TMR2RLL mov
  next c;

code noise-on ( - )
  $04 # TMR2CN mov \ Auto-reload, timer on.
  5 .IE setb \ Enable the T2 interrupt.
  7 .IE setb \ Enable global interrupts.
  next c;

code noise-off ( - )
  5 .IE clr \ Disable the T2 interrupt.
  $00 # TMR2CN mov \ Turn timer 2 off.
  next c;

: hz ( n - ) ;

: wpm ( - ) ;

Editor: morse.fs
```

The I/O pins of an f300 chip are configured as inputs by default after reset. The word “enable-sound” configures one port pin as a push pull output. This is the first “code” word we have encountered. AmrFORTH includes an 8051 assembler. The assembler is written in forth of course, and it uses RPN (Reverse Polish Notation) rather than algebraic notation, just as forth does. The definition for “enable-sound” has:

```
$04 # P0MDOUT orl \ pin 0.2 is push/pull output.
```

The \$ in \$04 means hexadecimal. The # means the immediate addressing mode. POMDOUT is the name of the Special Function Register that determines whether port 0 pins will be configured as inputs or outputs. We “orl” the \$04 bit mask in order to set the bit corresponding to pin 2 of port 0. Most code words end as this one does with “next c;” Next is just a macro for the RET instruction. “c;” ends a code definition just as “;” ends a colon definition, but it doesn't assemble any code itself. “next” is used instead of RET because this forth evolved from one where “next” was a macro that moved on to the “next” word to be executed. When we changed the internals of amrFORTH to compile native code, next became RET. It was easier to change the macro definition in one place than to change “next” to “ret” everywhere. The meaning is still the same, move on to the next word.

Next comes the timer2 interrupt routine. It is very simple. Whenever a timer2 overflow triggers this interrupt, the pin at P0.2 is toggled and the interrupt bit is cleared, followed by RETI, RETurn from Interrupt. As shown, the word “int!”, pronounced “int store” installs a jump to the interrupt routine at the timer2 interrupt vector, \$2b.

“steps” takes a number from the stack and stores it into the timer2 reload registers. Since timer2 counts upward to 65535 before overflowing to 0, the number of steps is inverted using the “cpl” instruction. In other words it is subtracted from 65535. This word will be the basis of “hz” later on.

The noise is turned on by properly configuring timer2 and then enabling its interrupt. The noise is turned off by disabling the interrupt and then turning off the timer.

This next screen shows how we determine steps from hertz. By steps we mean timer counts.

```
amrForth v6.1.0 28AUG03 /home/prog/Projects/morse
File Edit Search Compiler Programmer Options Help

Forth> Saving file: /home/prog/Projects/morse/morse.fs
Forth>

Interpreter: C8051F30x 24.5 MHz /dev/ttyS0 9600 interrupts-kernel

\ T2 runs at a rate of 24.5 Mhz divided by 12.
\ This means 24.5/12=2.0416666 us per step.
\ That equals 0.000002041666 sec per step.
\ Invert that to get 489796 steps per sec.
\ Frequency = cycles per sec = (steps/sec)/steps
\ Let n = number of steps per toggle, that's half a cycle.
\ freq = (489796/n)*2 (Multiply by 2 for a full cycle.
\ freq = 979592/n
\ n = 979592/freq
\ subtract n from 65535 and store that
\ in the timer reload registers.

: hz>steps ( n1 - n2)
  30 max 2000 min
  push 979592. pop um/mod nip ;

\ Specify the frequency of the sound in Hz.
: hz ( n - ) enable-sound hz>steps steps ;

Editor: morse.fs
```

The text above explains how we come up with the number 979592, here's how we use that number in forth. In `hz>steps` the first line clips the input value to the range 30 to 2000 hz. The number of hz is pushed onto the return stack temporarily so that the double precision (32 bit) number 979592 can be placed under it on the data stack. The period ending 979592. causes it to be interpreted as a 32 bit number. We pop the hertz value off the return stack and divide it into the 32 bit number already there. This is the number of timer2 steps required for the given number of hertz. The standard forth word `um/mod` divides an unsigned single (16 bits in amrFORTH) into an unsigned double, (32 bit here), returning an unsigned single remainder and an unsigned single quotient. The remainder is removed by `nip`, which drops the second number on the data stack.

Now when we compile and download the program we can test the code on the gadget. Do compile and download, then execute `' ' init' '` and `' ' noise-on' '` to see if it makes a noise. Of course this assumes that you have a piezo transducer connected across P0.2 and ground. We do hear the sound and can change the frequency using `' ' hz' '`, but we also notice that the word `' ' init' '` leaves a number on the data stack. This shouldn't happen. We look at the definition of `' ' init' '`:

```
: init ( - ) 1300 hz 5 wpm ;
```

and realize that wpm is just a stub that leave the 5 on the stack. This demonstrates that when you are making stubs you should be sure the stack effects make sense. We redefine “wpm” as follows:

```
: wpm ( n - ) drop ;
```

to preserve stack effects.

What we have now is the ability to turn a noise maker on indefinitely and then turn it off again. We need to be able to automatically turn off the noise maker after a given amount of time. That millisecond timer mentioned before is needed. The C8051F300 has another timer system called the PCA (Programmable Counter Array). It can be programmed to trigger an interrupt every millisecond which decrements a variable. We'll call the variable “wpm-counter”. Once the interrupt is running we can load the variable with a given number of milliseconds, and watch it count down to zero. When it reaches zero then we know that number of milliseconds has elapsed. The interrupt routine not only decrements the variable, but also increments the output compare registers to cause the next interrupt to occur in one millisecond. Here's the code:


```
amrForth v6.1.0 28AUG03 /home/prog/Projects/morse
File Edit Search Compiler Programmer Options Help
Forth> Saving file: /home/prog/Projects/morse/morse.fs
Forth>
Interpreter: C8051F30x 24.5 MHz /dev/ttyS0 9600 interrupts-kernel

code init-pca ( - )
    $08 # PCA0MD mov    \ Sysclk/1
    $49 # PCA0CPM0 mov   \ Software Timer (Compare) Mode
    $40 # PCA0CN mov     \ Start the timer.
    $08 # EIE1 orl      \ Enable pca interrupt.
    $80 # IE orl        \ Enable global interrupts.
next c;

\ 24.5 MHz = 24500 cycles per millisecond.
24500 $ff and constant ms-lo
24500 256 / $ff and constant ms-hi

cvariable wpm-counter

label pca-interrupt ( - )
    ACC push PSW push
    0 .PCA0CN set? if
        wpm-counter direct A mov
        0<> if wpm-counter direct dec then
            \ Set up for next interrupt in 1 ms.
            ms-lo # A mov PCA0CPL0 A add A PCA0CPL0 mov
            ms-hi # A mov PCA0CPH0 A addc A PCA0CPH0 mov
        then
        $40 # PCA0CN mov \ Clear interrupt bit.
        PSW pop ACC pop
        reti c;
pca-interrupt $4b int!

code ms ( c - )
    SP inc Apop A wpm-counter direct mov
    begin wpm-counter direct A mov 0= until
next c;

Editor: morse.fs
```

The values used in “init-pca” were determined by consulting the f300 data sheet from the Cygnal website. We configure PCA0 to use the straight system clock, at 24.5 Mhz. PCA0 is configured as a software timer, in compare mode. The timer is started running. The interrupt is enabled as are global interrupts. Next we define variables for the high and low bytes of the number 24500, the number of system clocks per millisecond. Those we call ms-hi and ms-lo. The variable “wpm-counter” is created. Note that it's a “cvariable”, that is, an 8 bit

variable. It will allow delays of up to 255 ms, which is sufficient. If the variable were 16 bits we would have to take care to disable interrupts when reading it to avoid reading a half updated variable. This is simpler.

The “pca-interrupt” begins by pushing the ACC and PSW registers. They will be changed and then restored by the interrupt routine since we are using addition. If the “wpm-counter” is not equal to zero we decrement it. This means that once it has counted down to zero it will stay there. Then the timer compare registers are incremented by 24500 in order to cause the next interrupt to occur in one millisecond. The interrupt bit is cleared, the registers restored, and then “reti”, return from interrupt. This routine is installed at the PCA interrupt vector at address \$4b.

Finally we can define “ms” easily. Though it could be defined in high level forth, it is simpler, faster, and more direct in assembler. The data stack in amrFORTH is in internal RAM, pointed to by SP which is an alias for R0. The data stack grows downward. “SP inc” ignores the high byte of the top of the stack, then the macro Apop moves the low byte into A, the accumulator, incrementing SP once more in the process. The value in A can be moved now into the variable, wpm-counter, and the program loops until the interrupt has counted “wpm-counter” down to zero.

When we compile, download, and interactively test this code we find that “ms” doesn' t work. No matter how long we tell it to wait it returns immediately. After rereading the code we find that we forgot to delete the old stub code for “ms”, which simply dropped the input value. This definition came after the new, real definition in the file, so it is the one we were executing. After erasing the stub, we find that “ms” seems to work. All we need is to define “wpm” to set the pacing, and we can actually hear some morse code.

The screenshot shows a window titled "amrForth v6.1.0 28AUG03 /home/prog/Projects/morse". It has a menu bar with File, Edit, Search, Compiler, Programmer, Options, and Help. The main area is divided into two panes. The top pane shows the Forth prompt and input: "Forth> -", "Forth> . - - .", and "Forth> duration c@ .". The bottom pane shows the interpreter status: "Interpreter: C8051F30x 24.5 MHz /dev/ttyS0 9600 interrupts-kernel". The main code editor pane contains the following text:

```
code ms ( c - )
    SP inc Apop A wpm-counter direct mov
    begin wpm-counter direct A mov 0= until
    next c;

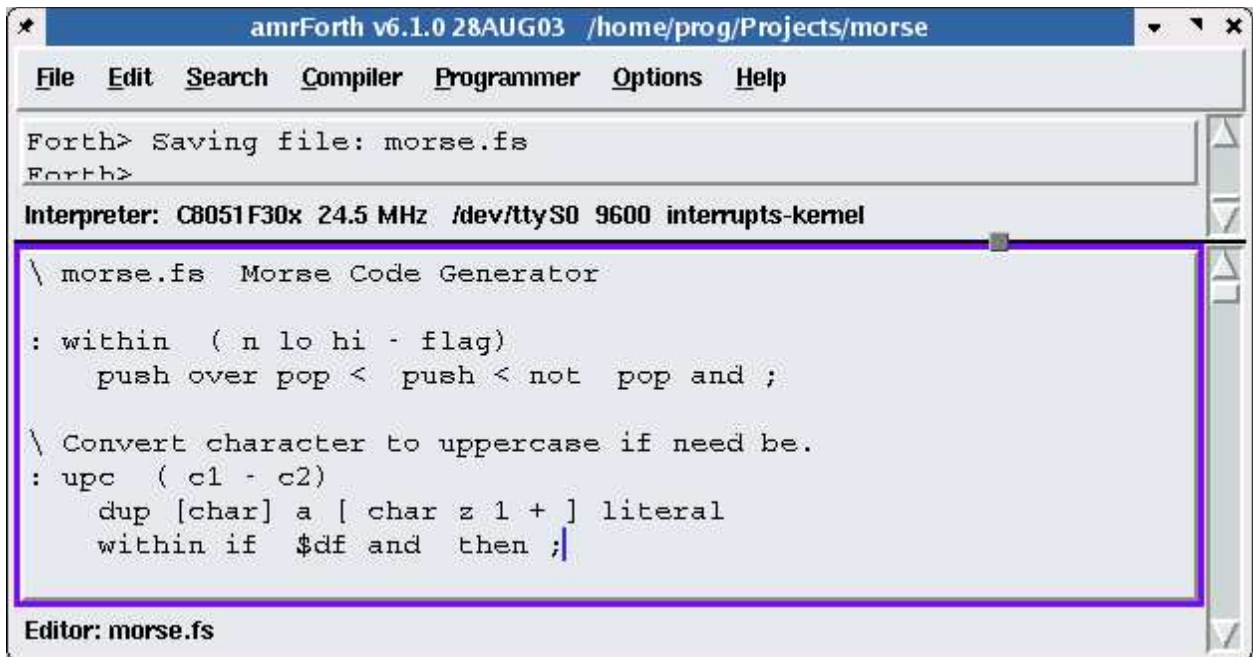
\ Using PARIS as the standard sized word,
\ there are 50 counts or space values per word.
\ ms = (1/((wpm*50)/60sec/min))*1000
\ ms = 1000/(wpm*50/60)
\ ms = 1000*60/wpm*50
\ ms = 1200/wpm

cvariable duration
: wpm ( wpm - ) 1200 swap / duration c! ;
```

At the bottom, it says "Editor: morse.fs".

As explained in the source code, we determine that the standard length of a “word” is 50 spaces and with algebra find that the duration of a space is equal to 1200 divided by the number of words per minute. This results in the definition of “wpm” shown above. Compile and download this program, and we can hear what each letter sounds like, except those whose names are already being used by the host, such as c and t. Try it, type “init” and then a b c. Oops, we heard a then b then the program was recompiled. “C” is a host command in forth mode. It runs the compiler. Likewise d, e, and t are host commands, which take precedence over target forth words. We could rename those words for testing purposes, but instead let’s just test “translate”.

First we want to convert characters to uppercase before feeding them to translate as written. The amrFORTH GUI is written in Tcl, which is not as flexible as we would like in some ways. If you type “char A” at the GUI forth> prompt, you will get the ascii code for lowercase a on the stack. We don’t know how to get around it at this point. Instead we use the word “upc” to convert to uppercase as the first thing done inside “translate”.



```
amrForth v6.1.0 28AUG03 /home/prog/Projects/morse
File Edit Search Compiler Programmer Options Help

Forth> Saving file: morse.fs
Forth>

Interpreter: C8051F30x 24.5 MHz /dev/ttyS0 9600 interrupts-kernel

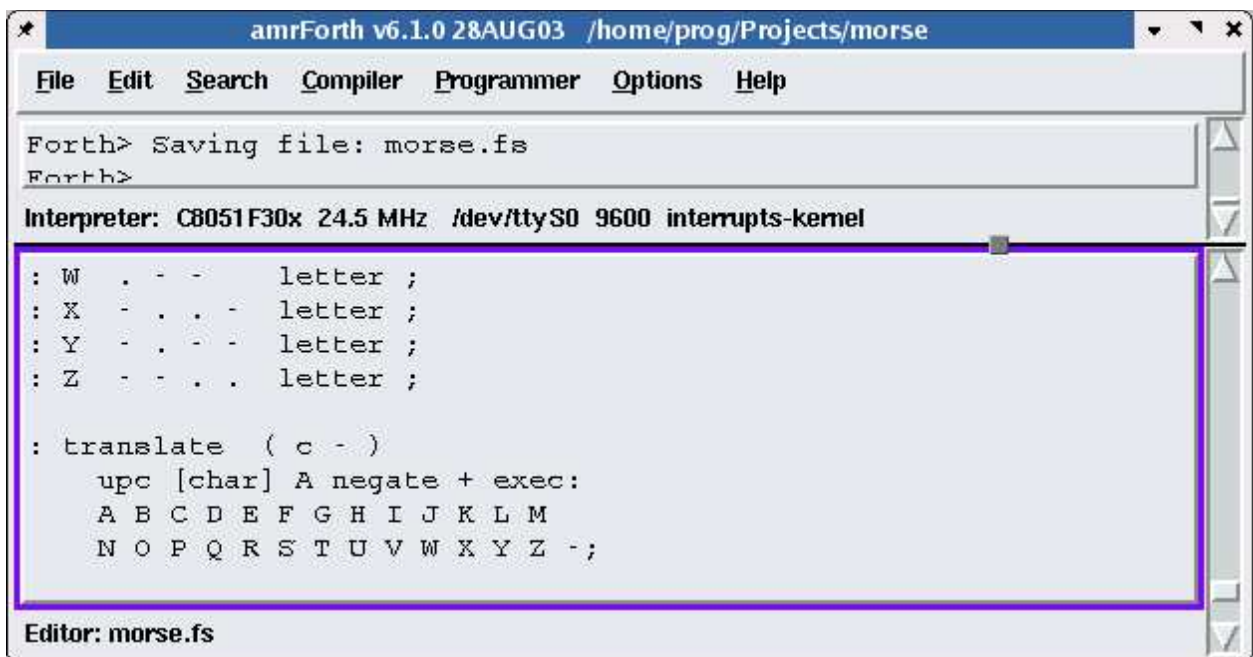
\ morse.fs Morse Code Generator

: within ( n lo hi - flag)
  push over pop < push < not pop and ;

\ Convert character to uppercase if need be.
: upc ( c1 - c2)
  dup [char] a [ char z 1 + ] literal
  within if $df and then ;

Editor: morse.fs
```

Within is a standard forth word that is not used often enough to be included in our default kernel. It returns a true or false flag if the number “n” in greater or equal to “lo” and also less than “hi”. “upc” changes a character to uppercase by clearing the lowercase bit in the ascii character code, but only if the original character was a lowercase character. Here' s what “translate” looks like now:



```
amrForth v6.1.0 28AUG03 /home/prog/Projects/morse
File Edit Search Compiler Programmer Options Help

Forth> Saving file: morse.fs
Forth>

Interpreter: C8051F30x 24.5 MHz /dev/ttyS0 9600 interrupts-kernel

: W . - - letter ;
: X - . . - letter ;
: Y - . - - letter ;
: Z - - . . letter ;

: translate ( c - )
  upc [char] A negate + exec:
  A B C D E F G H I J K L M
  N O P Q R S T U V W X Y Z - ;

Editor: morse.fs
```

After compiling and downloading you should be able to type “char a translate” at the command line and hear the morse code for A. If you happen to know the ascii character set of course you can specify the numeric code as well. For

example “\$41 translate” or “65 translate”. Each will result in the morse code for A. At this point you could even run the word “go” and talk to the gadget with a dumb terminal. There are a couple of problems with that still though. First the program will crash if you feed it any non-alphabetical character. Second, you need some way of changing the words per minute rate and the frequency in the final application.

Handling the non-alpha characters is easier, let' s do that first. What we need is a larger translation table, one that contains an action for every legal character, as well as a default error action for illegal characters. For now at least the action for illegal characters is to ignore them. After adding code for each legal morse code character and then adding those to the translation table, this is what you have:


```
amrForth v6.1.0 28AUG03 /home/prog/Projects/morse
File Edit Search Compiler Programmer Options Help
Forth>
Interpreter: C8051F30x 24.5 MHz /dev/ttyS0 9600 interrupts-kernel

: #0 - - - - - letter ;
: #1 . - - - - letter ;
: #2 . . - - - letter ;
: #3 . . . - - letter ;
: #4 . . . . - letter ;
: #5 . . . . . letter ;
: #6 - . . . . letter ;
: #7 - - . . . letter ;
: #8 - - - . . letter ;
: #9 - - - - . letter ;
: period . - . . . letter ;
: comma - - . . . letter ;
: colon - - - . . letter ;
: question . . - - . letter ;
: apostrophe . - - - . letter ;
: hyphen - . . . . letter ;
: fraction - . . . . letter ;
: paren - . - - . letter ;
: quote . - . . . letter ;
\ Default action for illegal characters.
: --- ( - ) ;

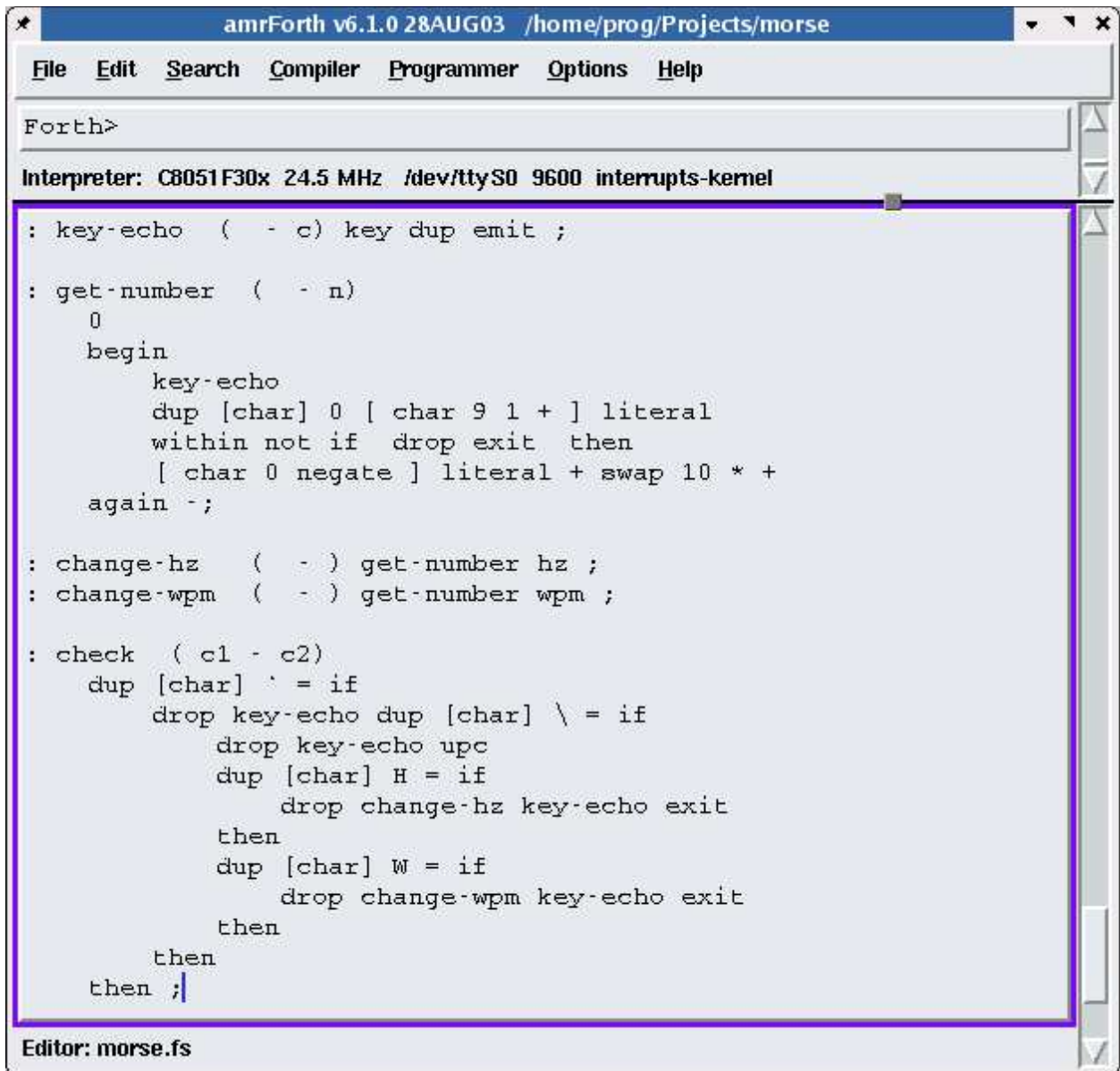
: translate ( c - )
  upc -31 + 0 max 60 min exec:
  --- word --- quote --- ---
  apostrophe paren paren --- ---
  comma hyphen period fraction ---
  #0 #1 #2 #3 #4 #5 #6 #7 #8 #9
  colon --- --- --- --- question ---
  A B C D E F G H I J K L M
  N O P Q R S T U V W X Y Z ---
  -;
```

Editor: morse.fs

Notice changes in the first line of “translate”. First we convert to uppercase using “upc” as before. Then we add a negative 31. Decimal 32 is the ascii code for the space character, which is the first ascii character we need to support. We subtract 31 instead of 32 so that we can clip all other control characters with 0 max and assign them the word ---, which does nothing. The space character produces a word space via “word”. The double quotes character executes “quote”, and so on. The table is constructed by consulting a table of

ascii characters. You should be able to find one in a book or on the internet. You could also write a simple forth program to make such a table for you. After changing “translate” you will need to test to be sure your table is correct. After compiling and downloading you might execute “char A translate A”, you should hear the same A character twice in a row if the table is correct. You probably will want to verify the table interactively using a dumb terminal, later, when the program is finished.

Now we can add some code to allow changing frequency and words per minute rate in the final application. What we'll do is add a word called “check” between getting the “key” and performing the “translation”. If “check” sees a special character sequence it will change the frequency or the rate. The special character sequence will start with “~”, sometimes called backquote. This character should never show up in an ascii character stream. The sequences “~hNNNNx” and “~wNNx” will stand for Hz and Wpm respectively. The “N” stands for any decimal numeric digit. The “x” stands for any non-numeric character.



```
amrForth v6.1.0 28AUG03 /home/prog/Projects/morse
File Edit Search Compiler Programmer Options Help
Forth>
Interpreter: C8051F30x 24.5 MHz /dev/ttyS0 9600 interrupts-kernel

: key-echo ( - c) key dup emit ;

: get-number ( - n)
  0
  begin
    key-echo
    dup [char] 0 [ char 9 1 + ] literal
    within not if drop exit then
    [ char 0 negate ] literal + swap 10 * +
  again -;

: change-hz ( - ) get-number hz ;
: change-wpm ( - ) get-number wpm ;

: check ( c1 - c2)
  dup [char] ' = if
    drop key-echo dup [char] \ = if
      drop key-echo upc
      dup [char] H = if
        drop change-hz key-echo exit
      then
      dup [char] W = if
        drop change-wpm key-echo exit
      then
    then
  then ;|

Editor: morse.fs
```

First “key-echo” is defined so that the dumb terminal will show what’s being typed. Then we need “get-number”, so that the user can enter a number for the new frequency or rate. This word loops while accepting characters over the serial line until a non-numeric character arrives. The numeric characters are accumulated into a number on the stack, and the non-numeric end character is discarded. Both “hz” and “wpm” were already defined earlier in the program. What “check” does is first check to see if the special character is on the stack. If it is then characters are accepted as long as they fit the pattern. If the complete pattern is filled out, the action is taken. If not the character is passed on to the word “translate” which comes after “check” in the final program (go).

After compiling and downloading this program should work with a dumb

terminal. There is still a problem though. Suppose you want to download a text file to the gadget and hear it all in morse code. The downloader will send characters too fast for the gadget to keep up. The serial buffer on the gadget is severely limited, only about 22 characters. We will solve this problem using X-ON, X-OFF flow control.

***** Discuss flow control here.