Max-FORTH V5.0L

This version of Max-Forth for the 68HC12 is modeled after Max-Forth V3.5E for the F68HC11 as closely as possible. The architecture of 68HC12 has influenced this realization. The main additions are making the fuzzy logic instructions available as Forth words and the addition of words to deal with flash memory much in the same way as the EEPROM words in V3.5.

Memory

Max-FORTH memory is a combination of internal and external RAMs and ROMs depending on which board you are using and how it is configured. On the NMIS/L-0012 there is:

1. internal registers at 0x00-0xFF
2. 1K of internal RAM at 0x800-0xBFF
3. 768 bytes of internal byte writeable EEPROM at 0xD00 to 0xFFF
4. 28K of external RAM at 0x1000 to 0x7FFF
5. 32Kbytes of internal Flash EEPROM at 0x8000-0xFFFF

The internal RAM is used for system variables and arrays. The Flash EEPROM contains Max-FORTH. The EEPROM and external RAM and the rest of the flash EEPROM are available for user programs. The high byte of a 16-bit value is in the low address of memory.

AutoStart

Autostart sequences can be placed at any $400 boundary starting at $400 if there are memory devices to support it. At bootup, the internal RAM is at $800 and can be used for an autostart test. The Max-FORTH autostart sequence is in high Flash followed by the kernel.

A simple test program to test autostart at $800:

```
COLD
: HI  ." Hello world" ;
HEX 800 AUTOSTART HI  ( set up autostart vector )
A44A 800 !            ( change it to a one shot )
```

Reset and you should get:

```
Hello world
Max-FORTH V5.0L
```

Using the control-g-reset sequence will skip the autostart.

There is also a check for an autostart in EEPROM which precedes the checks at the 400 boundaries. This is done because EEPROM has no locations at any of the autostart boundaries and it is consistant with the way the HC11 works. EEPROM starts at $D00 and an autostart can be placed there with the EE access words. Just like the HC11, the sequence A44A will signal that the autostart vector will run only once while A55A will signal that it is to be run continuously.

Quick Start (before autostart)

When the kernel first boots up, and before any write-once registers are written (such as COPCTL), a quick start sequence is checked for at EEPROM location FFE. This allows the COP to be turned on as a default. If it is not turned on, then the kernel turns it off.

A simple quick start routine which turns on COP (make sure there is an interrupt routine to handle it!!!!):

```
COLD
HEX
: COP-ON  7 16 C! ; EEWORD
A55A FFE EE!
' COP-ON CFA FFC EE!
```

Assembler and Indirect Threaded Code

The code threader uses two registers:

IP - instruction pointer; points within a list of word pointers
WP - word pointer; points to the current word to be executed

To thread, the word that IP points to is moved into the WP register and IP is incremented. Then an indirect jump through the word pointer starts executing the word's execution algorithm. The word's execution algorithm finishes by threading again.

IP is maintained in the D register unless it is being used to get a new WP. WP sits in the X register except for when it is first obtained it is in the D register. WP is only temporary and once it is not needed, then it can be used. It is used right at the beginning of the execution algorithm. To convert it to an IP it needs to be incremented from the cfa to the pfa.

In assembler, if X is needed as a temporary register, once WP has been used, it is not a problem. When the code word ends, then it must jump to the indirect code threader. If the code word needs D but not X then it transfers D to X and at the end, it jumps into a differen point within the indirect code threader. If both D and X are needed then D is pushed to the return stack. This is also used to nest threads.

FLASH and Kernel Reloading

The easiest way to download a new kernel is to type:

```
FLASH
```

and then erase and program. This avoids playing with jumpers. Just keep a jumper on J7 from GND to SHDN all the time to enable the erase voltage. The kernel is an S-Record. With the proper tools, code can be embedded within the kernel S-Record. This includes Forth, Assembler and C code. For more details, refer to our C-Inside product.

Flash and Program Storage:

Words can be stored into Flash just like the EEPROM on the 6811 with the restriction that a Flash location can only be used once. Unlike the EEPROM, the Flash can only be erased in bulk. This means that if you are storing code into Flash and then doing a COLD to remove it from the dictionary, unless you bulk erase the Flash and redownload the kernel, you'll have to choose a new location for your Flash code.

The equivelant words for Flashing code are:

```
FLC!  FL!  FLMOVE  FLWORD
```

If you are making extensive use of Flash then you can automate the free ROM area searching with the following tools. These tools automatically detect unused Flash memory and automatically put all definitions in Flash. The tools would typically be put at the start of a source file:

```
COLD
HEX
: Find-empty-space-in-Flash-ROM  ( -- a )
   F800  ( default)  F800  8000  ( limits)
   DO  FF  I 10 OVER + SWAP  DO I C@ AND LOOP  FF =  IF  DROP I  LEAVE THEN
   100 +LOOP  DUP F800 =  IF  CR ." Flash is full." CR  ELSE  DUP U.  THEN ;

Find-empty-space-in-Flash-ROM FDP !
FORGET Find-empty-space-in-Flash-ROM

( ==== ROM it all with auto-rom words ==== )
: ; [COMPILE] ; FLWORD ; IMMEDIATE  FLWORD
: CONSTANT  CONSTANT FLWORD ;
: CREATE  HERE CONSTANT ;
: VARIABLE  CREATE  2 ALLOT ;
```

The redefinition of ;, CONSTANT, CREATE and VARIABLE means that you don't have to sprinkle FLWORD throughout your code and all your code is automatically ROMed while the data space in RAM is managed as well.

EXRAM

On boards with external RAM available, like the NMIS/L-0012, the word EXRAM must be executed to make it visible. This word is only called automatically if there is already code in external RAM and a reset occurs.

RANDOM

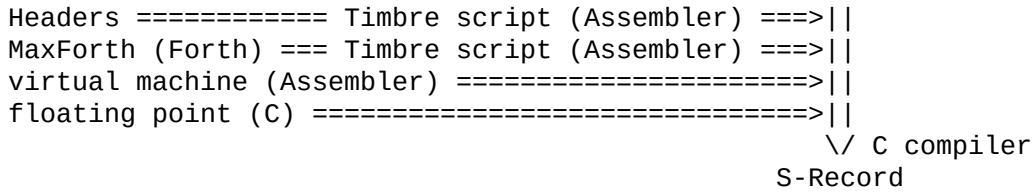
A random number generator has been added to the kernel. The variable "`seed`" contains the value used to calculate the next random number and can be set at the beginning of a program. RANDOM returns a random number between 0 and RAND_MAX.

Floating Point

As the kernel is maintained in high level source code for portability amongst different processors, the extensions are also in high level source code as well. Whereas the kernel is in Forth, the floating point is in C.

The floating point source code is linked with the kernel source code by compiling the two with the C compiler and producing an S-record output. The Forth source is compiled with a Timbre script to produce an assembler file. There is also a low level source file specific to each processor which defines the virtual machine.

Headers for the C words are defined in a separate file and translated to assembler by a Timbre script.



Fuzzy Logic Support

The fuzzy logic words give direct access and utilization in Forth to the fuzzy logic built into the processor. See the appendix in the CPU12 Reference Manual from Motorola for a good explanation of the fuzzy logic instructions.

What's Missing

There are a few items in Max-FORTH V3.5 which weren't, or haven't been, included in Max-FORTH V5.0. Most of the words missing are from the block word set. The support words for separated heads aren't included but the structure of the dictionary supports it. There is only one set of user variables and they are referred to as system variables.

FAQ

Q: My colleague has been helping me set up an HC12 to use an output compare to generate and handle an interrupt, but we do not know where the interrupt is vectored to. We read in the Motorola literature that an interrupt on Timer Channel 0 has a vector address of $FFEE-$FFEF. At this address is stored $FC65. This address is not writeable. At what address do I insert a JMP and address to my interrupt handler?

> A: There is a serial boot loader occupying the top 2K of memory which is locked against writing. All the interrupt vectors go through a routine which does an indirect jump through the same addresses but 2K lower. All you need to do is store your interrupt vector at that address, no jump instruction is required. For the timer channel 0, it will be at F7EE-F7EF. If you are programming this in Forth, you can do:
>
> HEX <address of interrupt routine> F7EE FL!

If you are programming in assembler or C, then just put the address of the interrupt routine at F7EE and download the S-record.

Q: I want to have my interrupt handler address located in EEPROM, so I redirect (again) program execution from, say F7EC (timer channel 1) to 0FE2 (EEPROM). Then in EEPROM I put a JMP hh ll to the interrupt handler, where hh ll are the high and low bytes of the address of the interrupt handler. For a simple test case, I have the following handler which doesn't work:

```
( a simple interrupt routine that writes "A" -ASCII 41- to $4100 )
HEX
CODE-SUB SIMPLEIRQ
86 C, 41 C,          ( 86 41    LDAA #$41 )
7A C, 41 C, 00 C,    ( 7A 4100  STAA $4100 )
0B C,                ( 0B       RTI )
END-CODE
```

the vector contents are:

```
location        contents
-----------------------
F7EC-F7ED       0F E2
0FE2-OFE4       06 17 CC     ( 17CC is the CFA of SIMPLEIRQ )
```

A: It should be the PFA, not the CFA. You can obtain this by:

```
' SIMPLEIRQ @
```

The CFA is just a pointer to a code routine. If you jump to it, then the pointer is executed by the processor as an instruction! The results are not predictable as you probably know.

Q+: We tried 17CC EXECUTE and 41 got written to $4100 as it should, although it hangs unless RTI is replaced with RTS.

A: This makes sense since it is declared as a CODE-SUB it must executed as a subroutine and the terminating instruction must be RTS. But when you execute it as an interrupt routine, you must use RTI as the return stack has different components on it (all the registers). What you would need to be able to execute it from Forth and from an interrupt would be a defining word called CODE-RTI which used a different inner interpreter.

Q+: The interrupt is generated by clearing the timer flags (FF TFLG1 C!), setting the interrupt mask (02 TMSK1 C!), clearing the I bit using a code routine analagous to that used on the HC11 and then inputting a pulse to the PT1 pin.

However, the above test did not work (processor hangs), so we simplified the scheme to simply a RTI at location OFE2. This failed to. We know that the interrupt is being generated, but it does not appear that execution gets to 0FE2. Have we missed something obvious?

A: When you generate an interrupt you must turn off the source of the interrupt. If you do not, then you will forever be interrupted. This is the situation in your case.

Q: When I down-load a Forth program to the HC12 any duplicate definitions causes the HC12 to reply with the name of the variable and " NOT UNIQUE". Is there a way to _not_ put this message on a new line? This is so that the download, which is waiting for a new-line, does not resume while the message is printing, thus causing the HC12 to miss characters and the download to crash. I seem to remember that duplicate names did not produce the same problems on the HC11.

A: I ran into that problem on the HC11. So I used this code to turn it off and actually speed up downloading. It also works on the HC12.

```
: NEW: LATEST PFAPTR LFA DUP >R @ 0 R@ ! >R : R> R> ! ;
NEW: : NEW: ; ( no more messages, not even for : )
```

Q: Are all the math routines in MAXforth reentrant, allowing them to be used in interrupt routines? If not, I will need to write some of my own. I don't really want to rewrite UM* and * if there's no need. I think I can assume that + , - , etc. are OK.

A: Here's the source for *, /MOD UM/MOD, UM*, + and -. Since they only use registers and not memory locations, they are interruptable. All the math operations are like this. The floating point is different, however. It uses two floating point locations in memory without saving the previous contents. So F* would not be reentrant and couldn't be used in an interrupt routine and a foreground routine unless it was redefined as:

```
: F* -INT  F*  +INT ;.
```

```
star::  .dw     .+2         ; code word
        pshd                ; save ip
        ldd     2,y+        ; pop top stack item
        ldx     0,y         ; get next stack item
        exg     x,y         ; setup for multiply
        emuls               ; signed multiply
        tfr     x,y         ; return stack pointer
        std     0,y         ; store answer to stack
        jmp     return      ; thread to next word

slash_mod::
        .dw     .+2         ; code word
        pshd                ; save ip
        ldd     2,y         ; get numerator
        ldx     0,y         ; get denominator
        idivs               ; signed 16/16 bit division
        stx     0,y         ; store quotient to top stack item
        std     2,y         ; store remainder as second stack item
        jmp     return      ; thread to next word

um_slash_mod:: ; ( d \ m -- r \ q )
        .dw     .+2         ; code word
        pshd                ; save ip
```

```
        ldx     2,y+        ; get denominator
        pshy                ; save stack pointer
        ldd     2,y         ; get lower numerator
        ldy     0,y         ; get upper numerator
        ediv                ; unsigned 32/16 bit division
        tfr     y,x         ; move quotient to x register
        puly                ; restore stack pointer
        stx     0,y         ; store quotient to top stack item
        std     2,y         ; store remainder as second stack item
        jmp     return      ; thread to next word

um_star::
    .dw     .+2         ; code word
        pshd                ; save ip
        ldd     0,y         ; get top stack item
        ldx     2,y         ; get next stack item
        exg     x,y         ; setup for multiply
        emul                ; unsigned 16,16,32 multiply
        exg     x,y         ; move upper to x register
        stx     0,y         ; store upper to stack
        std     2,y         ; store lower to stack
        jmp     return      ; thread to next word

; add top two values on the data stack
plus::  .dw     .+2         ; code word
        tfr     d,x         ; save ip into X
        ldd     2,y+        ; pop top value from data stack into D
        addd    0,y         ; add the next item to it
        std     0,y         ; store result on stack
        jmp     next        ; thread to next word

; subtract top from the next data stack item
minus:: .dw     .+2         ; code word
        tfr     d,x         ; save ip into X
        ldd     2,y         ; get second stack item
        subd    2,y+        ; subtract top item and drop
        std     0,y         ; store to stack
        jmp     next        ; thread to next word
```

Q+: I thought D was a free register; why save it? This is a push without a pop. "return" does the pop?

A: It has to do with the threading model on the 68HC12. On the 68CH11, two locations were used in memory for storing IP and WP during threading. On the HC12, those values are kept in registers. This has two advantages: speed and reentrancy. WP is kept in the X register and is only sometimes used at the beginning of a word (like a variable or constant). So the X register is available. IP is kept in the D register. This means that if the D register is needed in the word (typically math routines) and the X register is also needed, then the simplest thing to do is to push the D register. The jmp return takes it through code that pops the D register off the stack before threading on. In some cases the D register is not used and no adjustment is

required anyway. In a third category of assembler words, the D is needed and the X is available so then the D register is transferred to the X register.

For a more complete story, here's an excert from the kernel:

```
; Indirect threaded code machine
; The code threader uses two registers:
;   IP - instruction pointer; points within a list of word pointers
;   WP - word pointer; points to the current word to be executed
; To thread, the word that IP points to is moved into the WP register
; and IP is incremented.  Then an indirect jump through the word
; pointer starts executing the word's execution algorithm.  The
; word's execution algorithm finishes by threading again.
;
; IP is maintained in the D register unless it is being used to get
; a new WP.  WP sits in the X register except for when it is first
; obtained it is in the D register.  WP is only temporary and once
; it is not needed, then it can be used.  It is used right at the
; beginning of the execution algorithm.  To convert it to an IP it
; needs to be incremented from the cfa to the pfa.
;
; In assembler, if X is needed as a temporary register, once WP has been
; used, it is not a problem.  When the code word ends, then it must jump
; to the indirect code threader.  If the code word needs D but not X
; then it transfers D to X and at the end, it jumps into a different
; point within the indirect code threader.  If both D and X are needed
; then D is pushed to the return stack.  This is also used to nest
; threads.
;

; execute the word pointed to on the stack
execute::
        .dw     .+2         ; code word
        ldx     2,y+        ; get cfa off of stack
        jmp     [0,x]       ; thread to inner interpreter

; inner interpreter for macros built with : and ;
colon_ii::
        pshd                ; save current IP onto the return stack
        leax    2,x         ; change WP into new IP
copy1_of_next::             ; copied here for speed
        ldd     2,x+        ; get WP in D and increment IP
        exg     d,x         ; d is ip, x is wp, set for inner interpreters
        jmp     [0,x]       ; thread to an inner interpreter

; unthreader for : macros
exit::  .dw     .+2         ; code word
return::
        pulx                ; pull previous IP off the return stack
next::  ldd     2,x+        ; get WP in D and increment IP
        exg     d,x         ; d is ip, x is wp, set for inner interpreters
        jmp     [0,x]       ; thread to an inner interpreter
```

```
; restore IP and then thread for code words
itc::   tfr     d,x         ; get instruction pointer back into X
copy2_of_next:              ; copied here for speed
        ldd     2,x+        ; get WP in D and increment IP
        exg     d,x         ; d is ip, x is wp, set for inner interpreters
        jmp     [0,x]       ; thread to an inner interpreter

; inner interpreter for constant
cii::   movw    2,+x,2,-y   ; push constant value onto data stack
        bra     itc         ; thread to next word

; inner interpreter for variable
vii::   leax    2,x         ; get address of the variable
        stx     2,-y        ; push it onto the stack
        bra     itc         ; thread to next word

; inner interpreter for subroutines
sii::   pshd                ; save current IP onto the return stack
        jsr     2,x         ; call the appended subroutine
        bra     return      ; thread to next word

; inner interpreter for does children
dii::   pshd                ; save current IP
        leax    2,x         ; increment WP
        ldd     2,x+        ; get parent IP and increment WP
        stx     2,-y        ; push WP onto the data stack
        bra     itc         ; and thread

; innerinterpreter for C void functions(void)
fii::   sty     _dsp        ; save and publish stack pointer
        ldy     2,+x        ; put C function pointer into y
        tfr     d,x         ; get ip into X
        jsr     0,y         ; call C function
        ldy     _dsp        ; restore data stackpointer
        jmp     next        ; thread to next word
```

Q+: This isn't a machine instruction (nor are emul, idivs, or ediv below. I assume they are macros. Are they reentrant too?

```
        ldd     2,y+        ; pop top stack item
        ldx     0,y         ; get next stack item
        exg     x,y         ; setup for multiply
        emuls               ; signed multiply
```

A: On the 68HC12, emuls is an assembler instruction which takes two bytes for the opcode and 3 cycles to execute. So are the other math instructions. This also means that they are reentrant.

Q: I need to know the exact formula used by the max-forth about the floating-point stack. Some parameters will come by an external processor (intel) and i have to be sure that the communication protocol will transfer the exact term.

A: It is different on the HC11 and the HC12. For floating point on the HC11, 6 bytes are used. On the HC12, 4 bytes are used and it is the standard IEEE 32 bit single precision format. This'll be the same as used in C and consists of one sign bit, 8 bit exponent and a 23 bit mantissa.

Q: Can you tell me how to convert a number into a floating point value.

A: The best way (and most portable way) of dealing with floating point values is to use the provided translators:

D>F (d --)(F: -- r)
r is the floating-point equivalent of d.

F>D (F:r --)(-- d)
Convert r to d.

S>F (n--)(F: -- r)
r is the floating-point equivalent of n.

SF! (addr --)(F:r --)
Store the floating point number r as a 32 bit IEEE single precision number at addr.

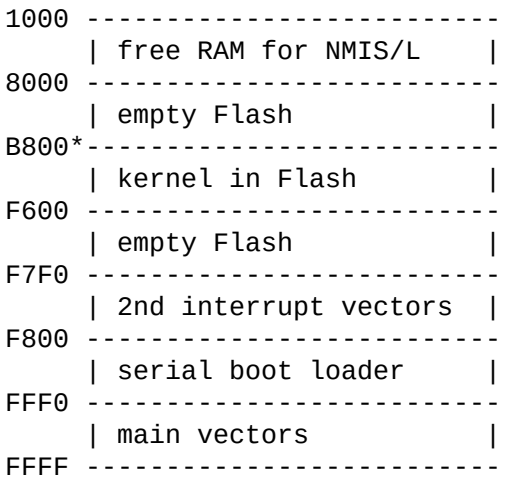
SF@ (addr --) (F: -- r)
Fetch the 32-bit IEEE single precision number stored at addr to the floating-point stack as in the internal representation.

Using these words you should be able to manipulate the values you require.

Appendix 1. Memory Map

This is the memory map for version 5.0L. The addresses with a star (*) beside them can change between releases.

```
0000 -------------------------
     | system registers       |
0100 -------------------------
     | no man's land          |
0800 -------------------------
     | autostart flag         |
0802 -------------------------
     | autostart vector       |
0804 -------------------------
     | kernel storage         |
0A0C*------------------------
     | free RAM (HERE)        |
0C00 -------------------------
     | no man's land          |
0D00 -------------------------
     | EEPROM                 |
```



Since the serial boot loader sits in high Flash ROM and it is write protected, the exception vectors which are at those locations have been replaced by a set of vectors at F7F0.

APPENDIX 2: System Variables

The system variables in release V5.0L are in this order and at these locations. Most of these are available from Forth. The location and order might change in future releases.

```
0976  r_0pfa
0978  s_0pfa
097A  fsp_0pfa
097C  dppfa
097E  lastpfa
0980  currentpfa
0982  outpfa
0984  fldpfa
0986  basepfa
0998  number_placespfa
099A  tibpfa
09EE  to_inpfa
09F0  blkpfa
09F2  number_tibpfa
09F4  spanpfa
09F6  statepfa
09F8  csppfa
09FA  warningpfa
09FC  uabortpfa
09FE  dplpfa
0A00  fencepfa
0A02  seedpfa
0A04  edppfa
0A06  edelaypfa
0A08  system_initializedpfa
0A0A  fdppfa
```

Appendix 3: WORDS

This is a listing of all the words in the dictionary that are displayed when WORDS is typed in and executed. The listing can be stopped at any time by pressing any key and then continued by hitting the space bar or the escape key to quit. This listing is from Max-FORTH V5.0L.

```
CB31 TASK           C4C0 (              B8BF @              B8D7 C@
B8CA !              B8E5 C!             C06D 2@             C065 2!
CCAF :              CCC1 ;              B8F4 +              B901 -
BF49 1-!            BF3F 1+!            BCA0 +!             BA2F *
BFAF /              BDBF ><             B88D SWAP           C08D 2OVER
C081 2SWAP          B899 DUP            BC3D 2DUP           BC23 OVER
BC51 ROT            C099 2ROT           C005 PICK           C039 ROLL
C013 -ROLL          B8A1 DROP           BC4A 2DROP          B8A7 >R
B8AF R>             BCF0 =              B956 NOT            BCDE 0=
C0A7 D0=            BF53 0>             B9AD 0<             BD0A U<
BCFA <              BE03 DU<            C077 D<             C0AF D=
B9C2 >              B929 AND            B938 OR             B947 XOR
CA8B IF             CA9B THEN           CAA7 ELSE           CAC1 BEGIN
CB01 UNTIL          CAED REPEAT         CADD WHILE          CACB AGAIN
CB13 END            CA19 DO             CA63 LOOP           CA77 +LOOP
B9EB K              B9E2 J              B9D9 I              B8B7 R@
CA2B LEAVE          B841 EXIT           BBCD KEY            BB9B EMIT
C4C9 ?TERMINAL      C05D S->D           BD3F ABS            C0C3 DABS
BD5E MIN            C0D1 DMIN           BD51 MAX            C0E5 DMAX
C1D4 SPACES         BFE1 DEPTH          C1B0 CR             C1EC TYPE
BD6B COUNT          C8B5 -TRAILING      B95F 1+             B97B 2+
B96D 1-             B989 2-             B90E 2/             B917 2*
BDF0 D+             BDDB D-             BDCE D2/            BA41 /MOD
BFB9 MOD            C10D */MOD          C119 */             BA66 UM*
BA51 UM/MOD         BC6F NEGATE         C0B7 DNEGATE        CCA1 CONSTANT
CCD3 VARIABLE       CD0D 2CONSTANT      CCDD 2VARIABLE      D3EC SF!
D3E8 SF@            D3AC FTAN           D3A4 FCOS           D3A8 FSIN
C759 FATAN2         D3DC FATAN          C2E4 F?             D3D8 FSQRT
D3A0 F2/            D39C F2*            CFBC F.S            C6F4 FNUMBER
C3C4 E.             C3BA F.             C39C (E.)           C373 (F.)
D38C F**            D37C FALOG          D3D0 FEXP           D394 2**X
D3D4 FLN            D390 FLOG           D3E4 LOG2           C79F ODD-POLY
C77F POLY           D3E0 FLOOR          D378 FROUND         C805 FLITERAL
D3C8 PI             D3CC e              C303 PLACES         D3B8 FLOAT+
D380 FLOATS         CCE7 FVARIABLE      CD01 FCONSTANT      CCF3 F,
D35C F!             D360 F@             D358 FABS           D3B4 FMIN
D3B0 FMAX           D374 F<             D398 F0<            D388 F0=
D3BC FNEGATE        D384 F>D            D354 S>F            D350 D>F
D34C F/             D348 F*             D344 F-             D340 F+
D36C FDROP          D368 FSWAP          D370 FOVER          D364 FDUP
D3C0 FNIP           BFF1 FDEPTH         D33C FSP            BFDD FSP0
BF01 TOGGLE         BA87 SP!            BA80 RP@            BA8F RP!
C50F UABORT         C50B WARNING        BFD5 R0             CBF2 SMUDGE
C821 DLITERAL       C531 MESSAGE        C579 ERROR          C5B3 ?ERROR
C5C1 ?COMP          C5D0 ?EXEC          C5DD ?PAIRS         C5E8 ?CSP
C603 ?STACK         BF33 @!             BC93 @@             B82E EXECUTE
BA79 SP@            BD93 CMOVE>         BD7A CMOVE          C4B8 ;S
```

```
CD2B CODE-SUB        CD1B CODE           CD3B END-CODE        CD5B USER
C2CC .               C2B6 .R             C2AC D.              C2D4 U.
C2C2 U.R             C290 D.R            C282 #S              C25F #
C24E SIGN            C23E #>             C234 <#              C2DC ?
C8E1 EXPECT          C958 QUERY          BFC1 BL              C4ED STATE
C151 CURRENT         C155 CONTEXT        C3D6 BLK             C121 DP
C204 FLD             C632 DPL            C3D2 >IN             C208 BASE
BFD9 S0              C3CE TIB            C3DA #TIB            C3DE SPAN
C3E2 C/L             C20C PAD            C125 HERE            C12D ALLOT
C141 ,               C135 C,             C1CC SPACE           BC62 ?DUP
C15B TRAVERSE        C1A4 LATEST         C9CF COMPILE         C4FD [
C4F1 ]               C210 HEX            C21B DECIMAL         CD67 ;CODE
CC73 <BUILDS         CC87 DOES>          C621 ."              CF6D .(
BDAC FILL            BFC5 ERASE          BFCD BLANK           C226 HOLD
C4A2 WORD            C670 CONVERT        C749 NUMBER          CB35 FIND
CC13 ID.             CC95 CREATE         C9DD [COMPILE]       C7F1 LITERAL
C851 INTERPRET       CBFD IMMEDIATE      C9E5 RECURSE         CA01 >MARK
C9F1 <MARK           CA0B >RESOLVE       C9F7 <RESOLVE        CD43 :CASE
C9C9 '               C9B9 [']            C175 LFA             C19E >BODY
C17B CFA             C183 NFA            C18F PFAPTR          CDE9 .LINE
D185 AUTOSTART       CB4D UNDO           CB6B FORGET          CE7C DUMP
CF7A .S              CEF1 WORDS          C989 QUIT            CB19 ABORT"
C51D ABORT           D17D COLD           BC0F BRANCH          B997 ?BRANCH
BEFD ATO4            D0F7 EEWORD         D0CD EEMOVE          D037 EEC!
D0B7 EE!             D01B EDP            BEA3 FUZZIFY
BEB9 EVALUATE-RULES  BEE5 DEFUZZIFY      BECB EVALUATE-WRULES D01F EDELAY
D2F0 FLWORD          D2C6 FLMOVE         D27A FLC!            D2B2 FL!
D25C FDP             BB28 FLASH          D143 EXRAM           CFF9 seed
CFFD RANDOM          CFF5 RAND_MAX       CB2D FORTH-83         OK
```

Appendix 4: Example Interrupt routine

This is an interrupt routine written in assembler used to increment a variable and read two A/D channels. It's compiled into the kernel using the C-Inside compiler. I've included an assembler listing. It could be easily modified to compile from Forth.

```
( ==== Real time interrupt ==== )
  VARIABLE ticks       ( incremented by RTI interrupt )

CODE SIR-RTI  ( real time interrupt: initiate a sensor reading
   BCLR  _rtiflg,$7F   ; clear RTIF
   JSR   read0123      ; read two A/D channels
   LDX   tickspfa      ; get timer contents
   INX                 ; increment it
   STX   tickspfa      ; store it back
   RTI                 ; return from interrupt

; interrupt vectors
.area exceptions(abs)
        .org     $F7F0
                .dw      sir_rti
.text
END-CODE
```

Generated assembler listing for the above code:

```
 D447                  sir_rti::

 D447  1D00157F           BCLR  _rtiflg,$7F   ; clear RTIF
 D44B  16D4BA             JSR   read01        ; read two A/D channels
 D44E  FE0A0C             LDX   tickspfa      ; get timer contents
 D451  08                 INX                 ; increment it
 D452  7E0A0C             STX   tickspfa      ; store it back
 D455  0B                 RTI                 ; return from interrupt
                              ; interrupt vectors
                              .area exceptions(abs)
                               .org     $F7F0
 F7F0 D447                           .dw      sir_rti
```

This is the code used to initialize it in Forth:

```
: INIT-A/D  ( -- )  -INT  ( disable interrupts )
   83 RTICTL C!      ( enable real time interrupt at 4ms )
   +INT ;  ( enable interrupts )
```

Where +INT and -INT are used to enable and disable interrupts:

```
CODE +INT
   .dw .+2             ; make callable
   cli                 ; enable interrupts
   jmp     itc         ; thread on
END-CODE

CODE -INT
   .dw .+2             ; make callable
   sei                 ; disable interrupts
   jmp     itc         ; thread on
END-CODE
```