

---

# **NMIN-2114**

## **Single Board Computer**

### **User Manual V.1**



# Table of Contents

<b>1.0 Overview</b> .....	<b>3</b>
1.1 Noted microcontroller features: .....	<b>3</b>
1.2 Included Files .....	<b>3</b>
<b>2.0 Getting Started</b> .....	<b>4</b>
<b>3.0 Memory Map</b> .....	<b>5</b>
<b>4.0 Programming the Board</b> .....	<b>5</b>
4.1 ONCE Connector and Parallel Port .....	<b>5</b>
4.2 S-Records and the Serial Loader .....	<b>6</b>
4.2.1 Downloading S-Records With X-Modem .....	<b>7</b>
4.2.2 Hooking Into Autoboot .....	<b>7</b>
4.2.3 Tags .....	<b>8</b>
4.2.4 Quick Entry .....	<b>8</b>
4.2.5 Boot Entry .....	<b>8</b>
4.2.6 Auto Vector .....	<b>8</b>
4.3 On-board Development System .....	<b>8</b>
<b>5.0 I/O Connections and Jumpers</b> .....	<b>8</b>
<b>6.0 Board Layout</b> .....	<b>10</b>
<b>7.0 Schematic</b> .....	<b>11</b>
<b>8.0 Examples</b> .....	<b>12</b>
8.1 Reading from the A/D port .....	<b>12</b>
8.2 Control Registers .....	<b>13</b>
8.3 Implementing an Interrupt .....	<b>14</b>
8.3.1 Setting up the CPU .....	<b>15</b>
8.3.2 Setting up the Interrupt Controller .....	<b>17</b>
8.3.3 Setting up a Peripheral Interrupt .....	<b>21</b>
8.3.4 Interrupts Calling Forth .....	<b>21</b>
8.4 Flash Programming .....	<b>22</b>
8.5 Auto Install Program .....	<b>25</b>
8.5.1 AUTOBOOT.F .....	<b>25</b>
8.5.2 UNBOOT2114.F .....	<b>27</b>
8.6 Advanced Programming .....	<b>27</b>
8.6.1 SRECBOOT.F .....	<b>28</b>
8.6.2 Example of C and Forth Together .....	<b>31</b>
8.6.3 Makefile .....	<b>34</b>
8.6.4 gnulink.acf .....	<b>34</b>
8.6.5 spi.c .....	<b>36</b>

8.6.6 pit.c . . . . .	<b>36</b>
8.6.7 forth.h . . . . .	<b>37</b>
8.6.8 micro.h . . . . .	<b>38</b>
8.6.9 interrupt.h . . . . .	<b>40</b>
8.6.10 words.c . . . . .	<b>40</b>
8.6.11 makewords.c . . . . .	<b>40</b>

## 1.0 Overview

The NMIN-2114 single board computer provides you with plug and play access to the powerful 32-bit Motorola MMC2114 microcontroller. The computer board provides power regulation, RS232 and RS422 serial support and an LCD connector. The microcontroller includes the following built in capabilities:

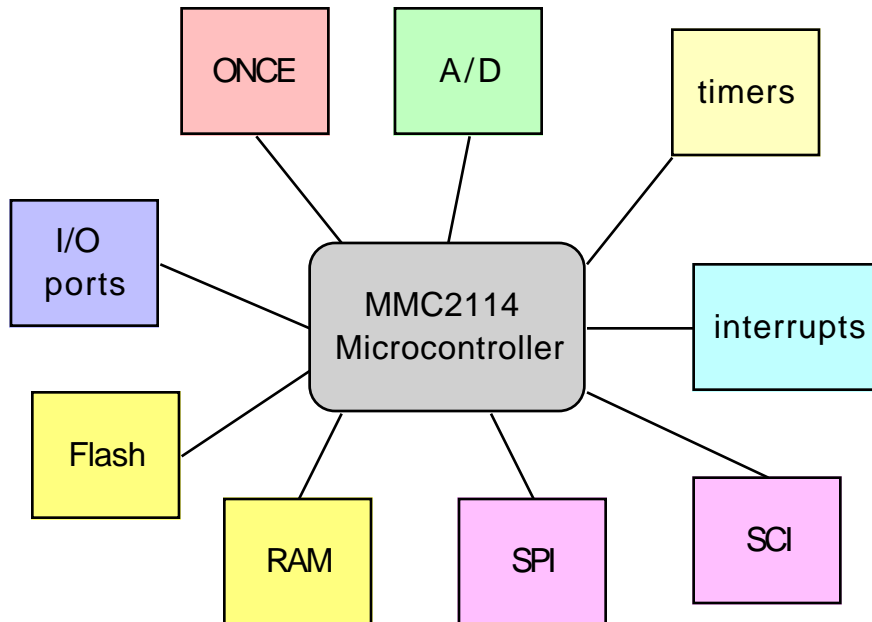


Figure 1 Peripheral interfaces on the 2114 microcontroller.

### 1.1 Noted microcontroller features:

- MCORE 32-bit RISC low-power integer processor
- 256K of word programmable flash memory
- 32K of static RAM
- 8 channels of 10-bit A/D with queueing
- 2 asynchronous serial channels
- 1 synchronous serial channel
- 8 timer channels with PWM capability
- up to 40 interrupts
- periodic interval timer
- watchdog timer
- 35 digital I/O pins
- eight external interrupts
- ONCE debug support

The computer board's power consumption with RS-422 drivers installed is about 80mA.

### 1.2 Included Files

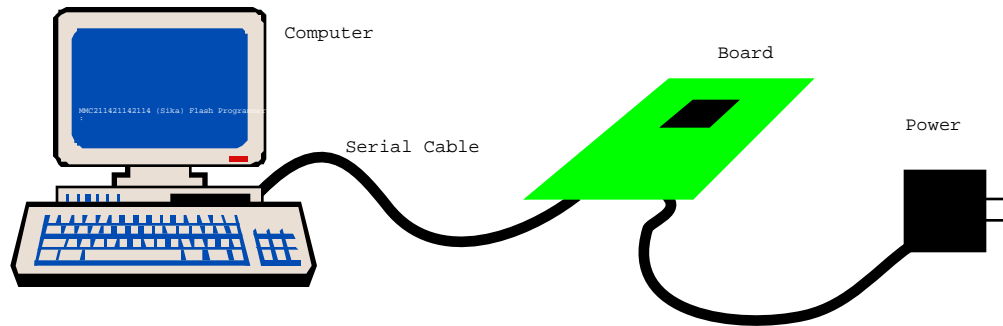
The following files are included and are available from our website. The MaxForth file is only available if you have licensed it.

- nmin2114v1.pdf - this manual in PDF format
- xload2114v2.s - serial loader file in s-record format that can be downloaded through the ONCE port or serial port
- mfcapp51b.s - MaxForth in s-record format
- srecboot.f - s-record loader and Forth loader which loads application programs into RAM or Flash
- autoboot.f - simple tool set for creating autostarting programs in flash

- unboot2114.f - for removing auto start tags

---

## 2.0 Getting Started



You will interact with your board by connecting it to a PC using a serial cable and running a terminal program such as HyperTerminal or an equivalent setup. You need to set it to 19200 BAUD, one stop bit, no parity and 8 data bits. Connect an RS232 serial cable between your PC's COM port and the serial port on the board. To power up the board, you need a 9 to 12 volt plug-in transformer plugged into the power jack, PJ1 (AC, DC both polarities accepted).

When everything is ready and you plug in the power, you should receive a prompt in the terminal program.

```
MMC2114 X-Modem Flash Programmer V.2
:
```

or

```
Max-FORTH V5.1B      (license agreement is required)
```

And when you depress the ENTER key, it should respond with

```
type ? for help
:
```

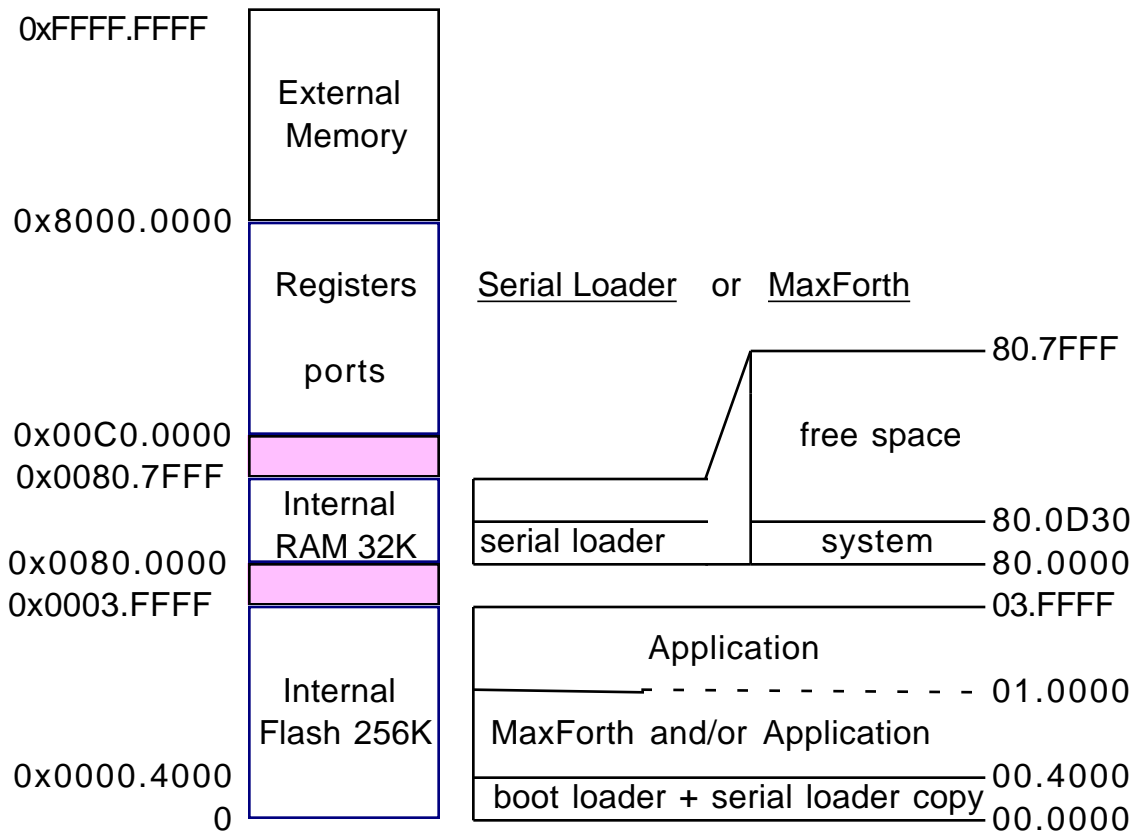
or

```
OK (Max-FORTH prompt)
```

When you see that message, it means the communication is established and you are ready to interact with the board and microcontroller. By pressing the reset button, SW1, you should get the same boot prompt as when you powered it up. Pressing the reset button will leave the contents of most of the RAM intact which might be useful for debugging purposes, whereas, if you power cycle the board, then all RAM contents will be lost.

### 3.0 Memory Map

The memory map consists of the RAM, ROM and registers. The interrupt vectors can exist anywhere by setting the vector base register (VBR) (Figure 5 on page 15).



**Figure 2** After booting, the RAM memory map either belongs to the serial loader or the MaxForth application.

### 4.0 Programming the Board

When the board is reset, it executes the program pointed to by the vector at location 0. In a loaded board this will be the boot loader which will either start the serial loader or MaxForth. When MaxForth starts, it checks to see if there is an application present and runs it or just runs MaxForth.

There are several ways in which to program the board:

1. download an s-record through the ONCE connector and the PC parallel port using a programmer such as CPROGMCZ.
2. download s-records using the embedded serial loader and a serial port with HyperTerminal
3. interact directly with the microcontroller and download source code to the on-board development system, MaxForth, through a serial port and HyperTerminal
4. download a mixture of s-records and Forth using the Forth s-record loader (SRECBOOT.F)

When downloading text files to the board, using the text download protocol, make sure the delay per line is at least 100 milliseconds. You can risk having lines missed if you go to fast, but with some setups, it is possible to use smaller line delays which has a nice effect on a long download. Your mileage will vary.

#### 4.1 ONCE Connector and Parallel Port

Using the MCore Cable from P&E Micro or equivalent, plug it into the parallel port of your computer

through a parallel port cable and the ONCE port on the computer board making sure to orientate the triangle on the pin header to pin 1 on the board which is marked by a square solder pad on the bottom of the board. Apply power to the board. When disconnecting from the board make sure it goes through a power cycle before you try out the downloaded software since a reset is not enough to regain control of the microcontroller after interacting with the ONCE port.

If you already have CPROGM CZ running, click reset the chip. If you start it up, it will reset the chip and proceed to the next choice which is to read in a configuration file. The file that works with the board is the 2114\_256k.MCP file which should be in the directory for configuration files. Next you will need to erase the module, specify an s-record to download and then program the module. To test out the software, you must disconnect the ONCE connector and power cycle the board. The serial loader file can be downloaded in this manner and then used to program the flash ROM using the serial port as described next.

Check the help screens for more details.

## 4.2 S-Records and the Serial Loader

Using the serial loader, you can download s-records that have been created by a C compiler or assembler system that you have acquired separately, to flash memory to be run. The help menu, invoked by typing a ?, is:

```
: ?
To download to flash:
  1. bulk erase if necessary
  2. Type f
  3. Send S-record with Xmodem or XModem-1K

f - Program flash
b - bulk erase all of the flash
e - erase application from flash
type ? for help
:
```

When programming the flash with an s-record, the locations to be programmed should be erased to 0xFF first by either erasing the whole flash or just the part which contains the serial loader.

### **WARNING:** Bulk Erase

*If you do bulk erase the entire flash with the 'b' selection, then the loader will be gone and you will not be able to reboot. Your application should either put a vector at location 0000, or you should download the serial loader program (xload2114v2.s).*

The serial loader works by running out of RAM. At bootup, the boot program in flash checks to see if there is an application at location 0x4000 by checking for a vector (anything but 0xFFFFFFFF). If there isn't, then it copies the serial loader program from flash to RAM and then runs the program. The serial loader program must run out of RAM to be able to program flash memory.

If you erase all of flash, then the next time you reset the board, there will be no programs present and you will have to program it through the ONCE port. Otherwise if you download a program with a reset vector at location 0x00000000, such as the serial loader program, then it will boot up next time. You have the flexibility to keep the serial loader program as part of your end system or to remove it and replace it with another program.

### **Quick Tip !!** Applications

If you are using the serial loader to test out your application which starts with a vector at location 0x4000, then just use 'e' and 'f' to download your application.

If you download an application, reboot and nothing happens, you can recover to the serial loader by shorting out PA7 to +3V on the J2 connector with a jumper or equivalent and pressing the reset button. You will be taken back to the serial loader where you can erase your errant application and try again.

### 4.2.1 Downloading S-Records With X-Modem

The serial loader uses the X-Modem protocol to transfer data from the host computer to the board. This allows for error recovery and pacing to be done by the protocol giving you a more robust download protocol. Depending on your host program and its defaults, the setting may be not right. Try it first and if it doesn't work, modify some of the settings.

To install a fresh copy of MaxForth and erase all the flash from 0x4000 to 0x1FFFF do the following:

1. get to the serial loader from Forth by FLASH or by resetting with PA7 to V+
2. press e
3. wait till done, then press f
4. download the MaxForth s-record with Xmodem
5. wait till done, then press reset with PA7 not connected to V+

### 4.2.2 Hooking Into Autoboot

You can autoboot an application by leaving a vector to it at location 0x4000. The startup boot loader will detect this vector and then jump to the location that the vector is pointing to. When MaxForth is installed, it has a vector at that location. To get to the serial loader from MaxForth, type in FLASH and hit enter. To get back to the serial loader from an application, execute the vector stored at location 0x190 in memory. The application area, 0x0.4000-0x1.FFFF can be erased using the serial loader without the boot loader being removed from memory. This is the 'e' option.

If MaxForth is loaded, you can check the memory at location hex 4000 by:

```
HEX 4000 1 DUMP
      0 1 2 3 4 5 6 7 8 9 A B C D E F
4000: 00 00 91 08 72 04 A4 02 30 04 60 03 1E 34 01 44 ...r...0.`..4.D OK
```

MaxForth occupies the space from 0x4000 to about 0xE300. At location 0x4000 is the start vector for MaxForth which is what the bootloader looks for when booting. If the location is 0xFFFFFFFF, then it is assumed that there is no application loaded and the serial loader is run. MaxForth can be replaced by erasing it and writing a new application in its place, making sure that the startup vector for the new application is at 0x4000.

If you want to hook into the Forth autoboot system, then there are several places where you can do this:

- quick entry - allow setting of COP and other write-once registers on the micro.
- boot entry - MaxForth has been setup and can be augmented
- auto vector - last possible chance before Forth is started up



**Table 1: Noted Memory Locations for Startup Hooks**

Name	Value	Notes
quick_tag	0x3FFC	store a tag <sup>1</sup>
quick_vector	0x3FF8	CFA of word to call
boot_tag	0x3FF0	store a tag <sup>2</sup>
boot_vector	0x3FF4	CFA of word to call
boot_start	0x10000	lower limit in flash for checking for an auto vector tag <sup>2</sup> on a 1K boundary
boot_end	0x1FFFF	upper limit of flash for auto vector tag <sup>2</sup> checking

1. A55A
2. A44A for first autostart or A55A for continuous last autostart.

### 4.2.3 Tags

The patterns 0xA44A and 0xA55A are referred to as tags and are used during the autoboot process to find vectors to be executed during the bootup process. Only the lowest A44A and A55A tag will be executed with A44A going first. This applies to the boot entry and autovectors. The quick entry, if used, only uses A55A.

### 4.2.4 Quick Entry

Quick entry lets you get in on the boot process and set COP or any other write once registers before MaxForth starts up. The tag contains a 32 bit value that is checked first and if present, then the CFA stored at the vector preceding it is executed.

```
A55A 3FFC FL! ' COP-RUN CFA 3FF8 FL! ( hook into quick entry vector )
```

### 4.2.5 Boot Entry

Boot entry lets you take over or execute something after MaxForth has been initialized. This is a good time to modify the dictionary linkage to add extra words from flash. The vector follows the tag.

```
A44A 3FF0 FL! ' STARTUP CFA 3FF4 FL! ( hook into boot-start vector )
```

### 4.2.6 Auto Vector

As well, at any 1K boundary in RAM you can lay down a tag followed by a vector.

```
A44A 1F800 FL! ' STARTUP CFA 1F804 FL! ( hook into auto-start vector )
```

## 4.3 On-board Development System

Taking advantage of the interactive nature of the board's development system, MaxForth, you can interact directly with the microcontroller's peripherals by fetching and storing values to the memory mapped configuration registers for the peripheral devices. This is an effective way of understanding the peripheral documentation, verifying correct initialization sequences, running some tests on different configurations and debugging driver code as you develop it. By typing in new definitions, you can add new macros to the dictionary for interactive use or for creating an automated program. Some examples are given later on.

---

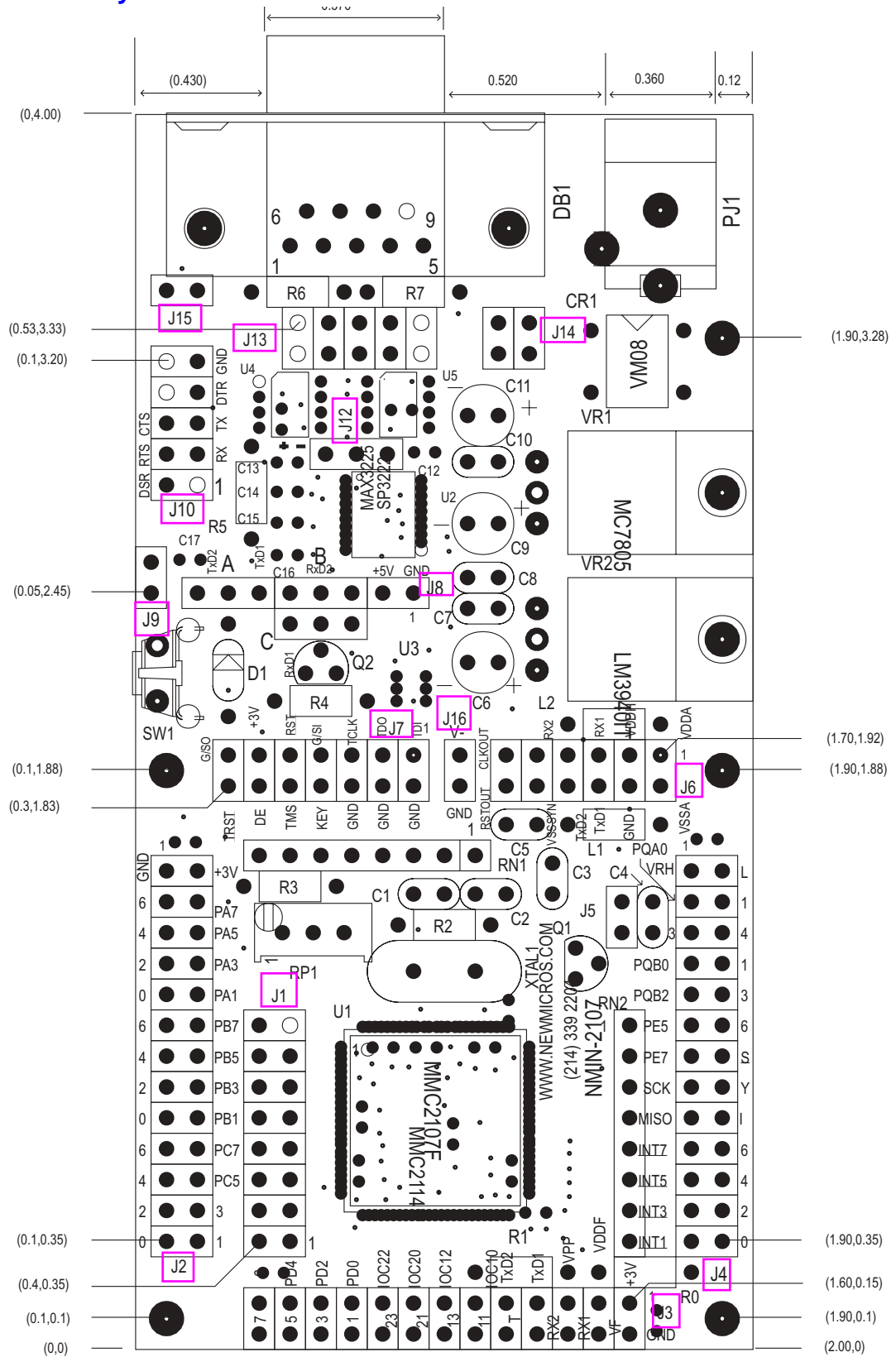
## 5.0 I/O Connections and Jumpers

The I/O connections J1-J16 are highlighted with colored boxes on the board layout on page 10. A description of their function follows:

- J1, LCD connector, match the triangle on the connector with the pin with a square solder pad.
- J2, J3, J4 are general purpose I/O connectors including timers, A/D, INT, etc.

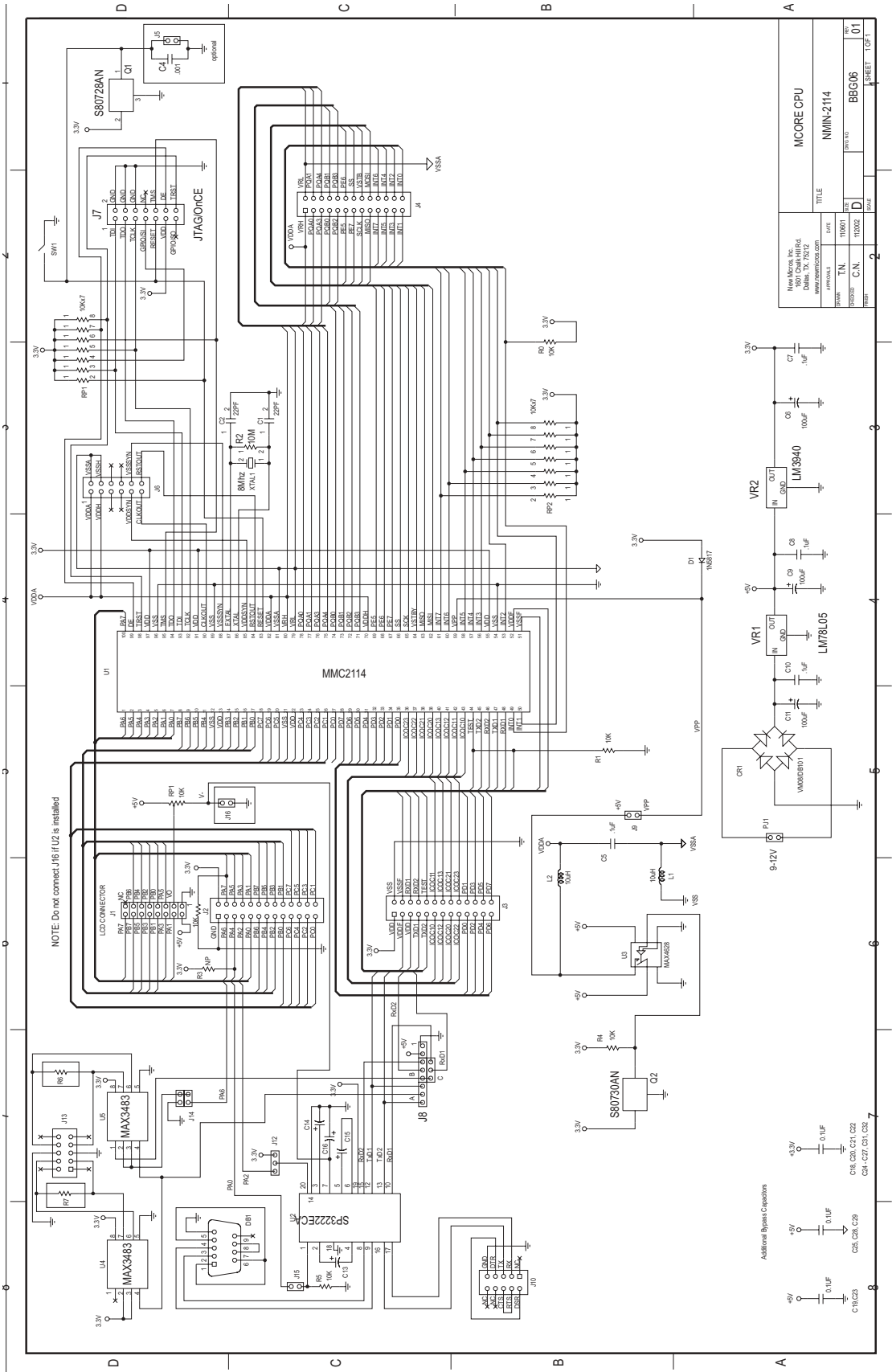
- J5, RESET & GND pin not installed. It may be used as option for a front panel mounted reset switch.
- J6, Misc. signals
- J7, JTAG/Once
- J8, includes jumper A, B, C. The Jumpers A,B,C allow both RS-232 drivers for COM1 & COM2 or 1 RS-232 and 1 RS-422 driver for either COM.
- Jumper A allows RS-422 selection of either TxD1 or TxD2 of SCI0 or SCI1 serial output signals respectively.
- Jumper B is the serial input to SCI1 labeled RxD2; it is defaulted to RS-232 but can be jumpered the other way for RS-422. (Next to Jumper B is +5V & GND. This can be used for a test point or external 5V source to power a probe. Use a paper separator for protection.)
- Jumper C is the serial input to SCI0 labeled RxD1. It is defaulted to RS-232 but can be jumpered for other way for RS-422.
- J9, Flash voltage supply. Connects when flash needs to Erase/Program. Open when flash does not need to erase/program or for Write protection.
- J10, serial connection for SCI1.
- DB1, serial connection for SCI0.
- J12, RS-232 shut down control input. Drive high for normal operation. Drive low to shut down the drivers. This can be controlled by PA2 or jumper to 3V for normal operation (default).
- J13, RS-422 connector
- J14, Jumper to GND (default) for RS-422 receiver always or jumper to PA6 as RS-485 transmitter/ Receiver control signal.
- J15 connects to pin 1 of RS-232, U2. This pin can be used to enable/disable the receivers for the RS-232 interface. For normal operation, this pin must be low. By default, J15 is open and this pin is pulled down through R5. To disable the receivers, J15 can be connected and controlled via PA0. Set it high to disable and low to enable the receivers.
- J16 is **only** needed when U2, the RS-232 chip, is not installed and the LCD application required. **Do not connect J16 if U2 is present.** This can damage U2.

## 6.0 Board Layout



Unit in Inches

# 7.0 Schematic



NewMicro, Inc. Dallas, TX 75212 www.newmicro.com		TITLE	
DATE	REV	DATE	REV
11/06/01	D	11/06/01	D
DESIGNER: C.N.	11/02/02	DESIGNER: BGC06	01
10/01	10/01	10/01	10/01
SHEET 2		SHEET 1 OF 1	

## 8.0 Examples

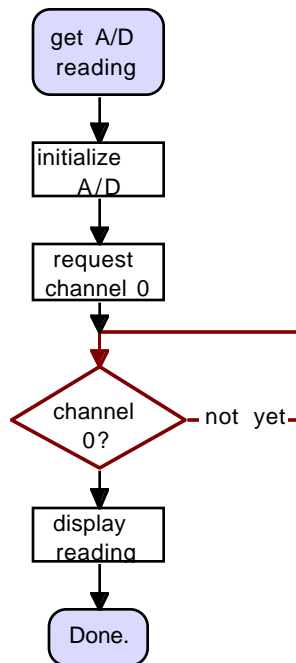
These examples may be typed in interactively or cut and pasted into the terminal window if you are using MaxForth. Alternately, they can be translated to the language that you are using to program the microcontroller with.

There are two manuals (PDF files from Motorola) that you need at your disposal to understand the 2114 microcontroller. Some of the subsequent tables and diagrams are from these sources:

- 1 **MMC2114 Technical Data**  
This document describes the peripherals, including the interrupt controller and the registers (memory-mapped, so accessible from Forth).
- 2 **MCORE Reference Manual**  
This document discusses the core CPU, instructions and internal registers (not memory mapped, so accessible only from assembler).

### 8.1 Reading from the A/D port

For this example we will consider the simplest way to get an A/D reading:



This involves: setting up the A/D registers so it is ready to go; requesting a read of a channel; waiting for that channel to complete converting; and finally reading and displaying the value. Each of these boxes on the diagram will become a word except for Done.

```
HEX
: H@ ( a -- h ) COUNT 100 * SWAP C@ OR ;
: DISPLAY-READING ( -- ) CA0280 H@ . ;
: CHANNEL0? ( -- f ) CA0010 C@ 80 AND ;
: CLEAR0 ( -- ) CHANNEL0? IF CA0010 DUP C@ 80 NOT AND SWAP C! THEN ;
: REQUEST0 ( -- ) CLEAR0 21 CA000C C! ;
: INIT-A/D ( -- ) 2 CA0200 C! C0 CA0201 C! 2 CA0202 C! FF CA0203 C! ;
: GETAD INIT-A/D REQUEST0 BEGIN CHANNEL0? UNTIL DISPLAY-READING ;
```

The first reading is from 5 volts, the second from 0 volts and the third from 3.3 volts all applied to pin PQB0 on connector J4.

```

GETAD 3D0 OK
GETAD 21 OK
GETAD 2A4 OK

```

In theory, 5 volts should be 3FF, 0 volts should be 0 and 3.3 volts should be 2A3. So while the limits are not quite there, the 3.3 volts is.

## 8.2 Control Registers

To control the CPU, you need to read and modify the contents of the control registers. Since the control registers are inside the CPU and not on the memory bus, you will need to access them through assembler code. The code below does this by using the `mtcr` and `mfcrr` assembly instructions ored with a specified control register number:

```

( MCore interface to internal registers Rob Chapman April 18, 2002 )
HEX

( Assembly interface: use mtcrr and mfcrr to transfer contents of control )
( registers to and from memory )
CODE MFCRR ( read control register into parameter field )
  ( lrw r4,ptr mfcrr r3,crn stl r3,(r4,0} jmp r15 ptr: .+4, 0, )
  74021003 , 930400CF , HERE 4 + , 0 , END-CODE

CODE MTCRR ( write control register from parameter field )
  ( lrw r3,value mtcrr r3,crn jmp r15 value: 0, )
  73021803 , 00CF0000 , 0 , END-CODE

( Forth interface: Create an opcode with the given control register )
( and then store it in the code word before running it. )
: CRN@ ( control register -- contents ) ( return contents of control reg# )
  10 * 74021003 OR ['] MFCRR @ >R R@ ! MFCRR R> C + @ ;

: CRN! ( value \ control register -- ) ( store value into control reg# )
  10 * 73021803 OR ['] MTCRR @ >R R@ ! R> 8 + ! MTCRR ;

CREATE MNEMS 13 4 * ALLOT ( pointers to 13 mnemonics )

: M" HERE SWAP 4 * MNEMS + ! 22 WORD C@ 1+ ALLOT ;
: ALIGN SP@ SP@ - ABS 1- DUP HERE + SWAP NOT AND DP ! ;

0 M" PSR " 1 M" VBR " 2 M" EPSR" 3 M" FPSR" 4 M" EPC " 5 M" FPC " 6 M" SS0 "
7 M" SS1 " 8 M" SS2 " 9 M" SS3 " A M" SS4 " B M" GCR " C M" GSR "
ALIGN

( Dump all the control registers )
: .CREGS ( -- ) CR D 0 DO I 2 .R I CRN@ 9 U.R
  I 4 * MNEMS + @ COUNT SPACE TYPE CR LOOP ;

```

PSR	CR0
VBR	CR1
EPSR	CR2
FPSR	CR3
EPC	CR4
FPC	CR5
SS0	CR6
SS1	CR7
SS2	CR8
SS3	CR9
SS4	CR10
GCR	CR11
GSR	CR12

**Figure 3 The control registers with their labels and values.**

This is the output from running .CREGS:

```
.CREGS
0 80000000 PSR
1      0 VBR
2      4984 EPSR
3 1DA83004 FPSR
4 8540380 EPC
5 4441280 FPC
6 28004023 SS0
7 220002C0 SS1
8 41406000 SS2
9 C26620 SS3
A 8200444 SS4
B      0 GCR
C      0 GSR
OK
```

### 8.3 Implementing an Interrupt

The 2114 contains a lot of parts to get right (usually all of them) before you can make an interrupt (more generally referred to as an exception) happen. You must set up the CPU, the interrupt controller and an interrupt source such as a peripheral. Setting up the CPU involves modifying CPU control registers while setting up the interrupt controller and peripheral involves modifying their memory mapped control registers.

The interrupt machinery on the 2114 supports a wide range of operational capabilities. You can just use one interrupt or support a complex system of prioritized interrupts from all peripherals. Interrupts can even be forced to happen to provide for a way of testing or syncing.

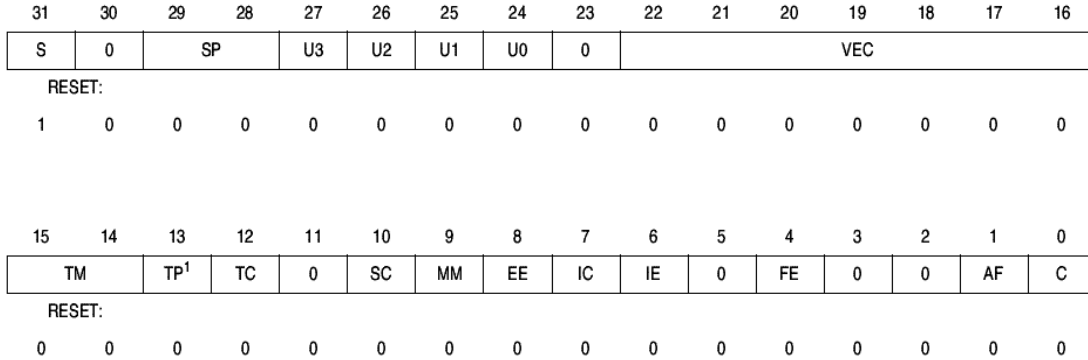
To set up an interrupt you'd follow these steps:

- 1 set up VBR and PSR (EE, IE)
- 2 set up vector for interrupt routine

- 3 set up interrupt controller (NIER)
- 4 set up peripheral

### 8.3.1 Setting up the CPU

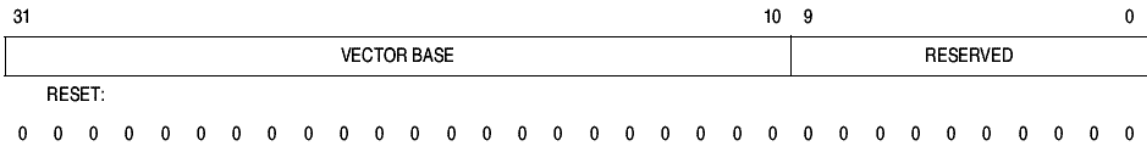
Setting up the CPU is a little harder because you need to access its internal registers (PSR and VBR) which can **only** be done in assembler.



**Figure 4 Program status register (PSR) accessible only from assembler. The EE and IE or FE bits must be set for interrupts to work.**

To enable interrupts at the CPU, you must set the EE (exceptions enabled) and one or both of the IE (interrupt enable) or FE (fast interrupt enable) bits. With the EE bit set, in the event of an interrupt, the PC (program counter) and PSR can be saved into two registers on the CPU. This limits interrupts to a depth of one unless some interrupt register management is added to push these two registers onto the stack and then restore them later. The IE bit enables normal interrupts while the FE bit enables fast interrupts.

```
80000140 0 CRN! ( enable exceptions and normal interrupts )
```



**Figure 5 The address in the VBR is the start of the exception vector table. Since the lower 10 bits are zero, the vector table can only start on address evenly divisible by 1024. 0, 0x400, 0x800...**

The VBR can be set to a RAM location for developing your interrupt routine and then later set to the final place for your working interrupt routine. You can put this near the top of the internal ram by:

```
807C00 1 CRN!
```

This puts the vector table in the upper 1K of the 32K internal RAM so the normal and fast interrupt vectors need to be placed at vectors 10 and 11 or 807C00 28 + and 807C00 2C +.



Vector Number(s)	Vector Offset (Hex)	Assignment
0	000	Reset
1	004	Misaligned access
2	008	Access error
3	00C	Divide by zero
4	010	Illegal instruction
5	014	Privilege violation
6	018	Trace exception
7	01C	Breakpoint exception
8	020	Unrecoverable error
9	024	Soft reset
10	028	$\overline{\text{INT}}$ autovector
11	02C	$\overline{\text{FINT}}$ Autovector
12	030	Hardware accelerator
13	034	(Reserved)
14	038	
15	03C	
16–19	040–04C	TRAP #0–3 instruction vectors
20–31	050–07C	Reserved
32–127	080–1FC	Reserved for vectored interrupt controller use

**Figure 6 The exception vector table pointed to by VBR. The 40 peripheral interrupts on the 2114 can be vectored to the 32-127 vectors or just through the normal and fast vectors.**

Your servicing interrupt routine must be in assembler and the vector that you place in the table must point to the start of the code. You must push registers if you are to use the same register file as normal processing and your routine must end with an `rti` instruction.



**NOTE:** Alternate register file

If you want to use the alternate file of registers (AF) for an interrupt, make sure you have the least significant bit in your interrupt vector set. The CPU masks off the lower two bits but also records the lsb into the AF bit enabling the alternate register file R0-15. Be careful though, if you try to use the stack pointer, it won't point to anything unless it has been initialized.

### 8.3.2 Setting up the Interrupt Controller

The interrupt controller is setup by modifying its memory mapped registers.

Address	Bits 31–24	Bits 23–16	Bits 15–8	Bits 7–0
0x00c5_0000	Interrupt control register (ICR)		Interrupt status register (ISR)	
0x00c5_0004	Interrupt force register high (IFRH)			
0x00c5_0008	Interrupt force register low (IFRL)			
0x00c5_000c	Interrupt pending register (IPR)			
0x00c5_0010	Normal interrupt enable register (NIER)			
0x00c5_0014	Normal interrupt pending register (NIPR)			
0x00c5_0018	Fast interrupt enable register (FIER)			
0x00c5_001c	Fast interrupt pending register (FIPR)			
0x00c5_0020 through 0x00c5_003c	Unimplemented <sup>(2)</sup>			
Priority level select registers (PLSR0–PLSR39)				
0x00c5_0040	PLSR0	PLSR1	PLSR2	PLSR3
0x00c5_0044	PLSR4	PLSR5	PLSR6	PLSR7
0x00c5_0048	PLSR8	PLSR9	PLSR10	PLSR11
0x00c5_004c	PLSR12	PLSR13	PLSR14	PLSR15
0x00c5_0050	PLSR16	PLSR17	PLSR18	PLSR19
0x00c5_0054	PLSR20	PLSR21	PLSR22	PLSR23
0x00c5_0058	PLSR24	PLSR25	PLSR26	PLSR27
0x00c5_005c	PLSR28	PLSR29	PLSR30	PLSR31
0x00c5_0060	PLSR32	PLSR33	PLSR34	PLSR35
0x00c5_0064	PLSR36	PLSR37	PLSR38	PLSR39
0x00c5_0068 through 0x00c5_007c	Unimplemented <sup>(2)</sup>			

**Figure 7** These registers are mapped into the memory space and can be accessed to set up the interrupt controller peripheral for interrupt processing. <sup>2</sup>Any access to the regions that are not implemented result in cycle termination errors

Setting up the interrupt controller and peripheral is a matter of translating documentation into which register bits to twiddle and since these are all memory mapped, you can access them from Forth. For a single interrupt to happen, we need to set NIER. For our purpose we have only one interrupt so we will not set its priority and leave it at 0. This means that we need to enable interrupts for priority level 0:

```


HEX
C50000 CONSTANT ICR ( interrupt control register
C50002 CONSTANT ISR ( interrupt status register
C50004 CONSTANT IFRH ( interrupt force register high
C50008 CONSTANT IFRL ( interrupt force register low
C5000C CONSTANT IPR ( interrupt pending register
C50010 CONSTANT NIER ( normal interrupt enable register

```

C50014 CONSTANT NIPR ( normal interrupt pending register  
 C50018 CONSTANT FIER ( fast interrupt enable register  
 C5001C CONSTANT FIPR ( fast interrupt pending register  
 C50040 CONSTANT PLSR ( base of priority level select registers 0-39  
 1 NIER !

For debugging, you can just fake interrupts using the interrupt controller force interrupt registers. The interrupt provides a way to create an interrupt for testing just by writing to a register. Since there are 40 possible interrupt sources from all the peripherals, two 32 bit registers are used to provide this: IFRH and IFRL. By setting a bit in these registers, you can force that interrupt from that peripheral to happen. In the interrupt service routine, you will need to reset that bit.

	Bit 31	30	29	28	27	26	25	Bit 24
Read:	0	0	0	0	0	0	0	0
Write:								
Reset:	0	0	0	0	0	0	0	0
	Bit 23	22	21	20	19	18	17	Bit 16
Read:	0	0	0	0	0	0	0	0
Write:								
Reset:	0	0	0	0	0	0	0	0
	Bit 15	14	13	12	11	10	9	Bit 8
Read:	0	0	0	0	0	0	0	0
Write:								
Reset:	0	0	0	0	0	0	0	0
	Bit 7	6	5	4	3	2	1	Bit 0
Read:								
Write:	IF39	IF38	IF37	IF36	IF35	IF34	IF33	IF32
Reset:	0	0	0	0	0	0	0	0

 = Writes have no effect and the access terminates without a transfer error exception.

**Figure 8 IFRH register contains the upper 8 interrupt source force bits.**

	Bit 31	30	29	28	27	26	25	Bit 24
Read:	IF31	IF30	IF29	IF28	IF27	IF26	IF25	IF24
Write:								
Reset:	0	0	0	0	0	0	0	0
	Bit 23	22	21	20	19	18	17	Bit 16
Read:	IF23	IF22	IF21	IF20	IF19	IF18	IF17	IF16
Write:								
Reset:	0	0	0	0	0	0	0	0
	Bit 15	14	13	12	11	10	9	Bit 8
Read:	IF15	IF14	IF13	IF12	IF11	IF10	IF9	IF8
Write:								
Reset:	0	0	0	0	0	0	0	0
	Bit 7	6	5	4	3	2	1	Bit 0
Read:	IF7	IF6	IF5	IF4	IF3	IF2	IF1	IF0
Write:								
Reset:	0	0	0	0	0	0	0	0

**Figure 9 IFRL with the first 32 interrupt force source bits.**

Source	Module	Flag	Source Description	Flag Clearing Mechanism
0	ADC	PF1	Queue 1 conversion pause	Write PF1 = 0 after reading PF1 = 1
1		CF1	Queue 1 conversion complete	Write CF1 = 0 after reading CF1 = 1
2		PF2	Queue 2 conversion pause	Write PF2 = 0 after reading PF2 = 1
3		CF2	Queue 2 conversion complete	Write CF2 = 0 after reading CF2 = 1
4	SPI	MODF	Mode fault	Write to SPICR1 after reading MODF = 1
5		SPIF	Transfer complete	Access SPIDR after reading SPIF = 1
6	SCI1	TDRE	Transmit data register empty	Write SCIDRL after reading TDRE = 1
7		TC	Transmit complete	Write SCIDRL after reading TC = 1
8		RDRF	Receive data register full	Read SCIDRL after reading RDRF = 1
9		OR	Receiver overrun	Read SCIDRL after reading OR = 1
10		IDLE	Receiver line idle	Read SCIDRL after reading IDLE = 1
11	SCI2	TDRE	Transmit data register empty	Write SCIDRL after reading TDRE = 1
12		TC	Transmit complete	Write SCIDRL after reading TC = 1
13		RDRF	Receive data register full	Read SCIDRL after reading RDRF = 1
14		OR	Receiver overrun	Read SCIDRL after reading OR = 1
15		IDLE	Receiver line idle	Read SCIDRL after reading IDLE = 1

**Figure 10 This table shows the first 16 interrupt sources, and how to turn them off.**

Source	Module	Flag	Source Description	Flag Clearing Mechanism
16	TIM1	C0F	Timer channel 0	Write C0F = 1 or access IC/OC if TFFCA = 1
17		C1F	Timer channel 1	Write 1 to C1F or access IC/OC if TFFCA = 1
18		C2F	Timer channel 2	Write 1 to C2F or access IC/OC if TFFCA = 1
19		C3F	Timer channel 3	Write 1 to C3F or access IC/OC if TFFCA = 1
20		TOF	Timer overflow	Write TOF = 1 or access TIMCNTH/L if TFFCA = 1
21		PAIF	Pulse accumulator input	Write PAIF = 1 or access PAC if TFFCA = 1
22		PAOVF	Pulse accumulator overflow	Write PAOVF = 1 or access PAC if TFFCA = 1
23		TIM2	C0F	Timer channel 0
24	C1F		Timer channel 1	Write C1F = 1 or access IC/OC if TFFCA = 1
25	C2F		Timer channel 2	Write C2F = 1 or access IC/OC if TFFCA = 1
26	C3F		Timer channel 3	Write C3F = 1 or access IC/OC if TFFCA = 1
27	TOF		Timer overflow	Write TOF = 1 or access TIMCNTH/L if TFFCA = 1
28	PAIF		Pulse accumulator input	Write PAIF = 1 or access PAC if TFFCA = 1
29	PAOVF		Pulse accumulator overflow	Write PAOVF = 1 or access PAC if TFFCA = 1
30	PIT1		PIF	PIT interrupt flag
31	PIT2	PIF	PIT interrupt flag	Write PIF = 1 or write PMR
32	EPORT	EPF0	Edge port flag 0	Write EPF0 = 1
33		EPF1	Edge port flag 1	Write EPF1 = 1
34		EPF2	Edge port flag 2	Write EPF2 = 1
35		EPF3	Edge port flag 3	Write EPF3 = 1
36		EPF4	Edge port flag 4	Write EPF4 = 1
37		EPF5	Edge port flag 5	Write EPF5 = 1
38		EPF6	Edge port flag 6	Write EPF6 = 1
39		EPF7	Edge port flag 7	Write EPF7 = 1

**Figure 11 This table has the rest of the interrupts from the other peripherals.**

If you want to develop a timer ticker to provide a solid time base you could use one of the PIT timers. To make sure you have the interrupt machinery right before you play with the PIT, you should test it with the interrupt force register. For PIT1 this is bit 30 or in hex:

```
40000000 IFR1 ! ( this forces on the interrupt bit for the PIT1 peripheral )
```

If we check out the interrupt controller registers, we can see that we have a normal interrupt pending in the ISR (lower 16 bits of ICR), the pending interrupt is lowest priority in the IPR register and it is a normal interrupt as shown by the value in NIPR:

```
ICR @ U. 80000200 OK
IPR @ U. 1 OK
NIPR @ U. 1 OK
```

At this point, we are ready to flip the switch on the CPU and test out an interrupt (unless you have done that already and crashed and burned). Before we switch on interrupts, we need to provide a vector for

the interrupt to jump to and also a way of turning off the source of the interrupt.

```
CODE TEST-PIT
( mvi r4,0 lrw r3,pIFRL stl r4,(r3,0} lrw r4,ctr )
60047303 , 94037404 ,
( addi r4,0 lrw r3,pctr stl r4,(r3,0} rte )
20047302 , 94030002 ,
( ptr: IFRL pctr: .+4 ctr: 0 )1
IFRL , HERE 4 + , 0 , END-CODE

: .TICK ['] TEST-PIT @ 18 + @ . ;
```

Now we set up a vector table and test the interrupt:

```
' TEST-PIT @ 1 OR 807C28 ! OK
1 NIER ! OK
807C00 1 CRN! OK
80000140 0 CRN! OK
.TICK 0 OK
40000000 IFRL ! .TICK 1 OK
40000000 IFRL ! .TICK 2 OK
```

Every time we force the interrupt on with IFRL, it gets serviced and the tick counter increments. Now we are ready to hook into a peripheral.

### 8.3.3 Setting up a Peripheral Interrupt

For this example, we will use the first programmable interrupt timer to generate a periodic interrupt which increments a variable. From the documentation we find that we need to set PCSR to run the PIT and the interrupt routine needs to write pit PIF to the PCSR to reset the interrupt. So our code will look like this:

```
CODE SIR-PIT
( mvi r4,13 lrw r3,pPCSR stb r4,(r3,0} lrw r4,ctr )
60D47303 , B4037404 ,
( addi r4,0 lrw r3,pctr stl r4,(r3,0} rte )
20047302 , 94030002 ,
C80001 , HERE 4 + , 0 , END-CODE

: .TICK ['] SIR-PIT @ 18 + @ . ;
```

Now we set up a vector table and test the interrupt:

```
' SIR-PIT @ 1 OR 807C28 ! OK
1 NIER ! OK
807C00 1 CRN! OK
80000140 0 CRN! OK
9FFFF C80000 ! OK
.TICK .TICK .TICK 31 36 3B OK
```

### 8.3.4 Interrupts Calling Forth

As for calling a Forth word from an interrupt, this is doomed to fail at some point since not all Forth words including the virtual machine are not interruptible without some extra context savings. This all adds overhead and goes against keeping interrupts as short as possible. If you keep your interrupt simple and

---

1 Pardon the hand coded horizontal assembler; it takes a while to get use to it. The next step would be to build a small assembler, throw away the code and compile the comments (horizontally)!

in assembler, then they have a greater chance of working and meeting system time constraints.

With the alternate register set, a parallel Forth environment which ran the same words and shared the same variables but had different stacks and program counters, could be built, in the sketch board of the mind. This parallel Forth could then be relegated to interrupts only and then you would be able to run one Forth from interrupts and one from the command line. Or better, there is just one Forth and it runs from interrupts only.

## 8.4 Flash Programming

On the 2114, there is 256K of programmable flash and 32K of RAM. The Flash can only be programmed by running programming software entirely out of RAM or from the opposite 128K block that is being programmed. The programming procedure is fairly simple and is expressed here as a C program:

```
// MCore 2114 flash routines  Rob Chapman  Nov 21, 2002

#include "mmc2114.h"

//SGFM Commands, User Mode
#define ERASE_VERIFY    0x05
#define PROGRAM_WORD    0x20
#define PAGE_ERASE      0x40
#define MASS_ERASE      0x41

#define FLASH_START     0x00000000
#define FLASH_END       0x0003FFFF
#define FLASH_SIZE      0x00040000

#define BANK_SIZE       0x00020000 //128k banks
#define SECTOR_SIZE     0x00002000 //8k sectors (minimum protectable area)

#define BANK0           0x00
#define BANK1           0x01

INT32U d,a,err; // global accessible data and address

void program(void) // program a word
{
    INT16U protect_mask;

    err = 0;
    protect_mask = ~(1<<((((a - FLASH_START) & (BANK_SIZE - 1)) / SECTOR_SIZE)));
    reg_SGFMPROT.reg = reg_SGFMPROT.reg & protect_mask;
    //clear the protects for the bank

    if (reg_SGFMPROT.reg != (reg_SGFMPROT.reg & protect_mask))
    //make sure that the protect got cleared (not locked)
    {
        err = 1;
        return;
    }

    if (reg_SGFMUSTAT.bit.CBEIF == 0) //make sure a command isn't active
    {
        err = 2;
    }
}
```

```

        return;
    }

    *(volatile INT32U *)a = d; //write the value to the address provided
    reg_SGFMCMDCMD.bit.CMD = PROGRAM_WORD; //write to the command buffer
    reg_SGFMUSTAT.bit.CBEIF = 1;
    //clear CBEIF flag by writing a one to start the command

    //wait for the command to complete
    while (reg_SGFMUSTAT.bit.CBEIF == 0);
    while (reg_SGFMUSTAT.bit.CCIF == 0);
} // so flash is available for use

void erase(void) // erase a page
{
    err = 0;

    //clear the protects for the bank
    reg_SGFMMPROT.reg = reg_SGFMMPROT.reg & 1;

    //make sure that the protect got cleared (not locked)
    if (reg_SGFMMPROT.reg != (reg_SGFMMPROT.reg & 1))
    {
        err = 3;
        return;
    }

    //clear error flags
    reg_SGFMUSTAT.bit.ACCERR = 1; //access error
    reg_SGFMUSTAT.bit.PVIOL = 1; //Protections Violation

    *(volatile INT32U *)a = d; //write the value to the address provided
    reg_SGFMCMDCMD.bit.CMD = PAGE_ERASE; //write to the command buffer
    reg_SGFMUSTAT.bit.CBEIF = 1; //clear CBEIF flag by writing a one to start command

    //wait for the command to complete
    while (reg_SGFMUSTAT.bit.CBEIF == 0);
    while (reg_SGFMUSTAT.bit.CCIF == 0);

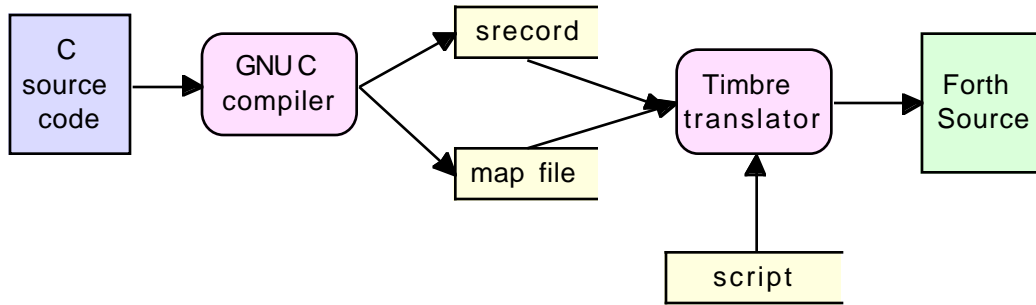
    //check for errors
    if (reg_SGFMUSTAT.bit.ACCERR == 1)
    {
        err = 4;
        return;
    }

    if (reg_SGFMUSTAT.bit.PVIOL == 1)
    {
        err = 5;
        return;
    }
}

```



Using the GNU C compiler to compile this C source into a map file and an S-record file and then passing



these two files through a Timbre script, we create a Forth compileable version:

```

( Flash interface; written in C and compiled to an srecord: flash2114.c
VARIABLE a    ( where to put flash address
VARIABLE d    ( where to put flash data
VARIABLE err  ( where error codes are stored  1-5 or 0

21C CONSTANT CLOCK
D00002 CONSTANT SGFMCLKD

DECIMAL

: INIT-FLASH ( -- ) ( initialize the flash CLKD register
  CLOCK @ 200000 /  DUP 60 >  IF  8 /  64 +  THEN  SGFMCLKD C! ;
INIT-FLASH

HEX
HERE ( program a 32 bit word with "d" at "a"
24709E00 , 743E6007 , 9704773D , 83072D17 , 16373ED7 , 60121225 , 1B7501F5 ,
01657639 , C7061657 , D706C706 , C6061657 , 0F76E802 , 9204F01F , 75358705 ,
37F7E003 , 60279704 , F0187732 , 87079703 , 76328706 , 2D8E35FE , 16E735D7 ,
97068705 , 35F79705 , 870537F7 , E0041256 , 870637F7 , EFFD7627 , 870637E7 ,
EFFD8E00 , 207000CF ,

HERE ( erase a page containing "a"
74216007 , 97047621 , C7062E17 , D706C706 , C6062E17 , 0F76E803 , 60379704 ,
00CF751C , 870535C7 , 97058705 , 35D79705 , 77178607 , 77198707 , 97067618 ,
87062D83 , 35F31637 , 35E79706 , 870535F7 , 97058705 , 37F7E004 , 12568706 ,
37F7E004 , 760E8706 , 37E7E004 , 760C8706 , 3FC72E17 , 2A17E003 , 60479704 ,
00CF8706 , 3FD72E17 , 2A17E002 , 60579704 , 00CF0000 ,
err ,          a , 00D00010 , 00D0001C ,          d , 00D00020 ,

CODE ERASE-PAGE  END-CODE  LATEST PFAPTR CFA !  ( connect to the C routines
CODE PROGRAM-WORD END-CODE  LATEST PFAPTR CFA !

```

This program consists of two parts: the part in the Forth user dictionary (the headers) and the C code which is in RAM. There is one word used to program and one word used to erase. Only a 1K page of memory can be erased at a time. In the 256K flash, there are 256 pages. Erasing leaves all the bits as FF. Flash can be programmed 4 bytes at a time aligned to a 4 byte boundary.

To program the flash, the location must be all FFs. You must store the data into the variable d and the destination address for flash must be in a. Once you set those two variables, then you call PROGRAM-WORD.

Erasing is about as simple. You must put the address of a location within the page into the variable a and then call ERASE-PAGE.

**Example:** Programming a word at 0x10000

```
HEX 10000 a ! ERASE-PAGE ( erase 10000-103FF )
12345678 d ! PROGRAM-WORD ( store 32 bit data at location 10000 )
```

You can check by:

```
10000 1 DUMP
```

## 8.5 Auto Install Program

This example uses the previous programming tool for flash to store Forth programs in empty flash. The program can be linked into the autostart system to initialize the dictionary and give a greeting.

There are two files involved:

AUTOBOOT.F - used to install a simple user program in flash that gets called at bootup

UNBOOT2114.F - this file can be used to remove all auto-starting tags.

If you download the first file, it will install the simple user program which will say Hello when restarted and add the word HI to the dictionary. If you download the UNBOOT2114.F file, it will cancel the autoboot feature from the simple user program.

### 8.5.1 AUTOBOOT.F

```
( Application tools Rob Chapman Aug 7, 2002 )
( put Forth application into Flash and allow hooking into autoboot )
COLD
HEX
: firstword ; ( dictionary sentry )

( Flash interface; written in C and compiled to an srecord: flash2114.c
VARIABLE a ( where to put flash address
VARIABLE d ( where to put flash data
VARIABLE err ( where error codes are stored 1-5 or 0

21C CONSTANT CLOCK
D00002 CONSTANT SGFMCLKD

DECIMAL

: INIT-FLASH ( -- ) ( initialize the flash CLKD register
CLOCK @ 200000 / DUP 60 > IF 8 / 64 + THEN SGFMCLKD C! ;
INIT-FLASH ( run so flash can be programmed

HEX
HERE ( program a 32 bit word with "d" at "a"
24709E00 , 743E6007 , 9704773D , 83072D17 , 16373ED7 , 60121225 , 1B7501F5 ,
01657639 , C7061657 , D706C706 , C6061657 , 0F76E802 , 9204F01F , 75358705 ,
37F7E003 , 60279704 , F0187732 , 87079703 , 76328706 , 2D8E35FE , 16E735D7 ,
97068705 , 35F79705 , 870537F7 , E0041256 , 870637F7 , EFFD7627 , 870637E7 ,
EFFD8E00 , 207000CF ,

HERE ( erase a page containing "a"
74216007 , 97047621 , C7062E17 , D706C706 , C6062E17 , 0F76E803 , 60379704 ,
00CF751C , 870535C7 , 97058705 , 35D79705 , 77178607 , 77198707 , 97067618 ,
```

```

87062D83 , 35F31637 , 35E79706 , 870535F7 , 97058705 , 37F7E004 , 12568706 ,
37F7EFFD , 760E8706 , 37E7EFFD , 760C8706 , 3FC72E17 , 2A17E003 , 60479704 ,
00CF8706 , 3FD72E17 , 2A17E002 , 60579704 , 00CF0000 ,
err ,          a , 00D00010 , 00D0001C ,          d , 00D00020 ,

```

```

CODE ERASE-PAGE END-CODE LATEST PFAPTR CFA ! ( connect to the C routines
CODE CPROGRAM-WORD END-CODE LATEST PFAPTR CFA !

```

```

: PROGRAM-WORD a @ IF CPROGRAM-WORD ELSE d @ . ." at zero " THEN ;

```

```

: FL! ( n \ a -- ) a ! d ! PROGRAM-WORD ;

```

```

: FLMOVE 4 / 0 DO >R DUP @ R@ FL! 4 + R> 4 + LOOP 2DROP ;

```

```

( Flash memory interface )

```

```

VARIABLE FDP ( flash dictionary pointer

```

```

VARIABLE flast ( points to last entry

```

```

( ==== Find empty space in Flash ROM ==== )

```

```

: FLBLANK ( -- a ) E400 40000 10000 ( aux \ high \ low )

```

```

DO FF I 40 OVER + SWAP DO I C@ AND LOOP ( must all be empty )

```

```

FF = IF DROP I LEAVE THEN

```

```

40 +LOOP DUP 1800 = IF CR ." Flash is full." CR ELSE DUP U. THEN ;

```

```

FLBLANK FDP ! UNDO ( use and discard )

```

```

: FLWORD ( -- ) LATEST >R ( remember the beginning of the word

```

```

FDP @ 3 + 3 NOT AND FDP ! ( set fdp to next word

```

```

R@ PFAPTR LFA @ CONTEXT @ ! ( unlink

```

```

['] firstword LFA @ R@ PFAPTR LFA ! ( relink below firstword

```

```

FDP @ ['] firstword LFA ! ( link to firstword

```

```

FDP @ R@ - R@ PFAPTR +! ( change pfa pointer

```

```

R@ FDP @ HERE R@ - FLMOVE ( move word

```

```

HERE R@ - FDP +! ( update flash pointer

```

```

R> DP ! ; ( update dictionary pointer

```

```

( ==== ROM it all with auto-rom words

```

```

: CONSTANT CONSTANT FLWORD ;

```

```

: CREATE HERE CONSTANT ;

```

```

: VARIABLE CREATE 2 ALLOT ;

```

```

: ; [COMPILE] ; FLWORD ; IMMEDIATE

```

```

( end of support code )

```

```

( user program begins here )

```

```

( ===== Test ===== )

```

```

( at reboot time a Hello will be emitted and the word that did it, will

```

```

( be part of the dictionary

```

```

: HI ." Hello" ;

```

```

HERE CONSTANT DP0

```

```

: STARTUP DP0 DP ! ['] HI NFA CONTEXT @ ! HI ;

```

```

( A44A 3FF0 FL! ' STARTUP CFA 3FF4 FL! ( hook into boot-start vector )

```

```
A44A 1F800 FL! ' STARTUP CFA 1F804 FL! ( hook into auto-start vector )
```

## 8.5.2 UNBOOT2114.F

```
( Utility for erasing boot tags in flash Rob Chapman Au072002  
( put flash programming tools in upper memory )  
COLD
```

```
HEX
```

```
( Flash interface; written in C and compiled to an srecord: flash2114.c  
VARIABLE a ( where to put flash address  
VARIABLE d ( where to put flash data  
VARIABLE err ( where error codes are stored 1-5 or 0
```

```
21C CONSTANT CLOCK  
D00002 CONSTANT SGFMCLKD
```

```
DECIMAL
```

```
: INIT-FLASH ( -- ) ( initialize the flash CLKD register  
CLOCK @ 200000 / DUP 60 > IF 8 / 64 + THEN SGFMCLKD C! ;  
INIT-FLASH ( run so flash can be programmed
```

```
HEX
```

```
HERE ( program a 32 bit word with "d" at "a"  
24709E00 , 743E6007 , 9704773D , 83072D17 , 16373ED7 , 60121225 , 1B7501F5 ,  
01657639 , C7061657 , D706C706 , C6061657 , 0F76E802 , 9204F01F , 75358705 ,  
37F7E003 , 60279704 , F0187732 , 87079703 , 76328706 , 2D8E35FE , 16E735D7 ,  
97068705 , 35F79705 , 870537F7 , E0041256 , 870637F7 , EFFD7627 , 870637E7 ,  
EFFD8E00 , 207000CF ,
```

```
HERE ( erase a page containing "a"  
74216007 , 97047621 , C7062E17 , D706C706 , C6062E17 , 0F76E803 , 60379704 ,  
00CF751C , 870535C7 , 97058705 , 35D79705 , 77178607 , 77198707 , 97067618 ,  
87062D83 , 35F31637 , 35E79706 , 870535F7 , 97058705 , 37F7E004 , 12568706 ,  
37F7EFFD , 760E8706 , 37E7EFFD , 760C8706 , 3FC72E17 , 2A17E003 , 60479704 ,  
00CF8706 , 3FD72E17 , 2A17E002 , 60579704 , 00CF0000 ,  
err , a , 00D00010 , 00D0001C , d , 00D00020 ,
```

```
CODE ERASE-PAGE END-CODE LATEST PFAPTR CFA ! ( connect to the C routines  
CODE CPROGRAM-WORD END-CODE LATEST PFAPTR CFA !
```

```
: PROGRAM-WORD a @ IF CPROGRAM-WORD ELSE d @ . ." at zero " THEN ;
```

```
: FL! ( n \ a -- ) a ! d ! PROGRAM-WORD ;
```

```
: ?UNBOOT ( a -- ) DUP @ DUP A44A = SWAP A55A = OR  
IF DUP . 0 SWAP FL! ELSE DROP THEN ;
```

```
: UNBOOTS ( clear out the boot flags in 3FF0 and 3FFC and any in Flash  
3FF0 ?UNBOOT 3FFC ?UNBOOT 1FBFF 10000 DO I ?UNBOOT 400 +LOOP ;
```

```
UNBOOTS
```

## 8.6 Advanced Programming

This loader file which is written in Forth, runs on top of MaxForth and allows you to mix S-Records and Forth programming in the same file. This is useful if you want to program exception vectors in C or

assembler and then debug them with Forth. S-records of S0, S1, S2, S3, S7, S8 and S9 formats are accepted. For an S7, S8, or S9 record, the vector of the start program will be run at the time the line is downloaded. This can be used to initialize the program, add words to the dictionary or to run a sequence.

The file SRECBOOT.F, is installed on top of MaxForth and becomes a resident program in Flash which is called on MaxForth bootup. The MaxForth prompt will be replaced with:

```
Boot V.3
```

```
*
```

The asterick (\*) will be displayed after each non-blank line has been received and processed. A line could be an s-record or a line of Forth code. If you type in Forth code, it will not be echoed but it will execute. You can test this by trying .S. If you want to get back to the Forth prompt, just type in QUIT and hit enter.

Once you download the program and reboot, the loader will be resident and running waiting for code to be downloaded. You can remove the program at anytime by typing in -AUTOBOOT and hitting the enter key.

### 8.6.1 SRECBOOT.F

```
( S-Record loader and application support  Rob Chapman  Nov 21, 2002 )
( This small program consists of the following tools:
( > an s-record loader for S0, S1,S2,S3 and S7,S8,S9 records for RAM
( > any line not starting with S will be passed to the Forth interpreter
( > a code to FLASH saver/restorer with autoboot
( > saving and restoring static ram images
( > application auto start

( === From Reset to application running
( reset MCU
( jumps to MCore bootloader routine
( calls MaxForth startup routine
( scans for secondary boot and finds image in FLASH

COLD
HEX
( ==== Image and properties ==== )
  HERE CONSTANT RAM-START  ( start of user dictionary
 10000 CONSTANT FL-START   ( start of Flash above Forth
 1BFFF CONSTANT FL-END    ( just before forth vector at 1FC00

( Flash interface; written in C and compiled to an srecord: flash2114.c
VARIABLE a    ( where to put flash address
VARIABLE d    ( where to put flash data
VARIABLE err  ( where error codes are stored  1-5 or 0

21C CONSTANT CLOCK
D00002 CONSTANT SGFMCLKD

DECIMAL

: INIT-FLASH ( -- ) ( initialize the flash CLKD register
  CLOCK @ 200000 /  DUP 60 >  IF  8 /  64 +  THEN  SGFMCLKD C! ;
INIT-FLASH

HEX
```

```

HERE ( program a 32 bit word with "d" at "a"
  24709E00 , 743E6007 , 9704773D , 83072D17 , 16373ED7 , 60121225 , 1B7501F5 ,
  01657639 , C7061657 , D706C706 , C6061657 , 0F76E802 , 9204F01F , 75358705 ,
  37F7E003 , 60279704 , F0187732 , 87079703 , 76328706 , 2D8E35FE , 16E735D7 ,
  97068705 , 35F79705 , 870537F7 , E0041256 , 870637F7 , EFFD7627 , 870637E7 ,
  EFFD8E00 , 207000CF ,

```

```

HERE ( erase a page containing "a"
  74216007 , 97047621 , C7062E17 , D706C706 , C6062E17 , 0F76E803 , 60379704 ,
  00CF751C , 870535C7 , 97058705 , 35D79705 , 77178607 , 77198707 , 97067618 ,
  87062D83 , 35F31637 , 35E79706 , 870535F7 , 97058705 , 37F7E004 , 12568706 ,
  37F7EFFD , 760E8706 , 37E7EFFD , 760C8706 , 3FC72E17 , 2A17E003 , 60479704 ,
  00CF8706 , 3FD72E17 , 2A17E002 , 60579704 , 00CF0000 ,
  err ,          a , 00D00010 , 00D0001C ,          d , 00D00020 ,

```

```

CODE ERASE-PAGE  END-CODE  LATEST PFAPTR CFA !   ( connect to the C routines
CODE CPROGRAM-WORD END-CODE  LATEST PFAPTR CFA !

```

```

: PROGRAM-WORD  a @  IF  CPROGRAM-WORD  ELSE  d @ . ."  at zero "  THEN ;

```

```

: FL!  ( n \ a -- )  a ! d !  PROGRAM-WORD ;

```

```

: FLMOVE  4 / 0  DO  >R  DUP @  R@ FL!  4 +  R> 4 +  LOOP  2DROP ;

```

```

( ==== Startup services ==== )

```

```

R0 DPL MIN      CONSTANT sys-start   ( start of system variables
R0 DPL MAX 4 + CONSTANT sys-end      ( end of system variables
  sys-end sys-start - CONSTANT sys-size  ( size of variables
( + 0  0000A44A
( + 4  RESTORE - must point to saved image of restore
( + 8  main startup word; must be a PFA
( + C  copy of sys-size RAM locations in system
( + C + sys-size  length of image
( +10 + sys-size  image of code

```

```

: FLERASE  ( -- )  ( erase FLASH
  FL-END FL-START  DO  I a !  ERASE-PAGE  400 +LOOP ;

```

```

: FL+  FL-START + ;

```

```

: >FL  ( ram -- flash )  ( convert ram address to saved location in flash
  RAM-START - FL-START + 10 + sys-size + ;

```

```

: RESTORE  ( can't call nonexistent words so [ ] LITERAL is used
  [ C FL+ ] LITERAL          ( start of system image )
  [ sys-start ] LITERAL      ( start of system variables )
  [ sys-size ] LITERAL  CMOVE ( restore system RAM image
  [ sys-size 10 + FL+ ] LITERAL ( start of dict image )
  [ RAM-START ] LITERAL      ( start of dict space
  [ sys-size C + FL+ ] LITERAL @  CMOVE ( restore dictionary image to RAM
  [ 8 FL+ ] LITERAL @
  DUP -1 =  IF  DROP [' ] TASK @ THEN
  >R ; ( run main program after exiting restore

```

```

: SEEMOVE  ( s \ d \ n -- )  ( like FLMOVE but you get to see it

```

```

100 /MOD DUP
IF SWAP >R DUP >R 0
    DO I 1+ . 2DUP 100 FLMOVE 100 + >R 100 + R> LOOP
    R> R> SWAP
THEN 1+ . FLMOVE ;

: FLSAVE ( -- ) ( save application image in flash )
    FLERASE ( always start flash the same way.
    A44A FL-START FL! ( fire once autostart flag for startup )
    ['] RESTORE CFA >FL 4 FL+ FL! ( restoration vector )
    ( ['] TASK @ 8 FL+ FL! ( null vector can be written by INSTALL )
    sys-start C FL+ sys-size FLMOVE ( save system variables )
    RAM-START sys-size 10 + FL+ HERE RAM-START -
    DUP sys-size C + FL+ FL! SEEMOVE ( save length & dictionary ) ;

: INSTALL ( tick -- ) FLSAVE @ 8 FL+ FL! ;

: -AUTOBOOT 0 FL-START FL! ( turn off autostart flag ) COLD ; ( cold boot

( ==== record parser
: X 0 ( a \ c -- a' \ n )
    SWAP 0
    DO 10 * >R COUNT
    30 - DUP
    9 > IF
        7 - THEN R> + LOOP ;

( ==== Memory interfaces ==== )
VARIABLE accumen ( accumulate 4 bytes then write
VARIABLE storage ( point to where it is to be stored

: WRITE-OUT ( -- ) storage @ 40000 U< ( flash write? )
    IF accumen @ -1 XOR ( skip if nothing set; else program flash )
        IF storage @ 3 NOT AND a ! accumen @ d ! PROGRAM-WORD THEN
        ELSE accumen @ storage @ ! THEN ( RAM write )
    0 storage ! -1 accumen ! ; ( initialize values again )

: BYTE! ( b \ a -- ) DUP storage @ XOR 3 NOT AND ( same block of 4? )
    IF WRITE-OUT THEN DUP 3 NOT AND storage ! 3 AND accumen + C! ;

: S! ( t \ a \ c -- t' ) 0 DO >R 2 X R@ BYTE! R> 1+ LOOP DROP ;

( ==== S record loader
VARIABLE ls ( last srecord
VARIABLE g ( go address from s-record

' TASK CFA g ! ( default

: G g EXECUTE ;

: Q ( -- )
    BEGIN TIB @
        BEGIN KEY DUP BL < 0= WHILE OVER C! 1+ REPEAT DROP ( echoless )
        0 >IN ! DUP TIB @ - DUP SPAN ! #TIB ! 0 OVER C! ( imitate query )
        TIB @ XOR UNTIL ; ( skip blank lines )

```

```

: 0INPUT ( -- ) WRITE-OUT 0 DUP >IN ! DUP #TIB ! TIB @ C! ;

: Szero ( a -- a' ) 2 X 2* X DROP ;

: S1 ( a -- a' ) 2 X 3 - >R 4 X R> S! ;
: S2 ( a -- a' ) 2 X 4 - >R 6 X R> S! ;
: S3 ( a -- a' ) 2 X 5 - >R 8 X R> S! ;

: S7 ( a -- a' ) WRITE-OUT 2 X DROP 8 X g ! 0INPUT ;
: S8 ( a -- a' ) WRITE-OUT 2 X DROP 6 X g ! 0INPUT ;
: S9 ( a -- a' ) WRITE-OUT 2 X DROP 4 X g ! 0INPUT ;

: S ( -- )
  BEGIN ." *" Q
    TIB @ 2 X
    DUP 1C0 = IF DROP Szero DROP ELSE
    DUP 1C1 = IF DROP S1 DROP ELSE
    DUP 1C2 = IF DROP S2 DROP ELSE
    DUP 1C3 = IF DROP S3 DROP ELSE
    DUP 1C7 = IF 1s ! S7 ." *" EXIT ELSE
    DUP 1C8 = IF 1s ! S8 ." *" EXIT ELSE
    DUP 1C9 = IF 1s ! S9 ." *" EXIT
              ELSE 2DROP INTERPRET
    THEN THEN THEN THEN THEN THEN THEN
  AGAIN ;

( ==== Half word support ==== )
: H@ ( a -- n ) COUNT 100 * SWAP C@ OR ;
: H! ( n \ a -- ) >R 100 /MOD R@ C! R> 1+ C! ;
: H, ( n -- ) HERE 2 ALLOT H! ;

( ==== Boot program ==== )
: START -1 accumen ! INIT-FLASH CR ." Boot V.3" CR
  S 1s @ 1C6 > IF G THEN ;

808000 HERE - DECIMAL CR . .( bytes of memory left.) HEX

' START INSTALL

```

## 8.6.2 Example of C and Forth Together

This is an example of a program which has interrupt handlers written for SPI and PIT in C but the control interface is in Forth:

```

( MCore high speed SPI receiver for sensors Rob Chapman Nov 12, 2002 )
: \ 0 WORD DROP ; IMMEDIATE \ allow comments like this

```

```

\ PIT and SPI exception handlers written in C

```

```

S00A00007069742E7331399D
S31A00806000760D870620079706770C60069607740B6067B7047535
S31A008060150B6017B705770AB607B60564C73477B704664639162D
S31A0080602A870635279706000200CF0080721C0080722000CB0099
S30E0080603F0000CB000800CB00072D
S31A008060487709A70776097509A7051C677609A606B607A605616F

```



```

S31A0080605D170C67E80312672007B705000200CF00CB00030080B8
S30F0080607272240080722000CB000526
S31A0080607C24F0007D122E123D7F29014276298306752987059795
S31A008060910612263476B6072007970512272407127201422C872E
S31A008060A60F72E80D12748705A60EB607200E2007970512272418
S31A008060BB07127201420F42E7F4751B87052407A6073476B607F5
S31A008060D087052E372A07E809125660058706B507200797062E1A
S31A008060E5372A07E7F9751187059307203797051276207696077E
S31A008060FA87051276203696059D1720779705006D20F000CF24AF
S31A0080610F70007E123E73087F08760687069E0720379706006EA5
S31A00806124207000CF008061AC008005080080050C000048880006
S3080080613980607C81
S31A0080613C760EA7062007B70600CF24709F00720B730C7F0C72B8
S31A008061510D73087F0D720D730E7F0B720E730E7F0A720E730F89
S31A008061667F08720F730F7F078F00207000CF00807220008061AD
S31A0080617BEC0080613C0080607C008061F40080610E008061F887
S31A0080619000807224008062000080721C0080620C008060480058
S30C008061A58062140080600097
S31A008061AC122512272E372A07E808F0012002A7022A07E7FC058D
S31A008061C15200CF2032860277071C671F67740616472A07EFF7D3
S31A008061D6F0012002A7022A07E7FC055200CFEFEFEFEFF808080BF
S306008061EB80AD
S31A008061EC68656C6C6F000000696E640073616D706C6573007361
S31A00806201616D706C655F6E6F0000007369725F737069007369E2
S30B00806216725F70697400DE
S70500806146D3

```

HEX

```

: .SPI ind C@ . ;

( Assembly interface: use mtcr and mfcr to transfer contents of control )
( registers to and from memory )
CODE MFCR ( read control register into parameter field )
( lrw r4,ptr mfcr r3,crn stl r3,(r4,0} jmp r15 ptr: .+4, 0, )
74021003 , 930400CF , HERE 4 + , 0 , END-CODE

CODE MTCR ( write control register from parameter field )
( lrw r3,value mtcr r3,crn jmp r15 value: 0, )
73021803 , 00CF0000 , 0 , END-CODE

( Forth interface: Create an opcode with the given control register )
( and then store it in the code word before running it. )
: CRN@ ( control register -- contents ) ( return contents of control reg# )
10 * 74021003 OR ['] MFCCR @ >R R@ ! MFCCR R> C + @ ;

: CRN! ( value \ control register -- ) ( store value into control reg# )
10 * 73021803 OR ['] MTCR @ >R R@ ! R> 8 + ! MTCR ;

CREATE MNEMS 13 4 * ALLOT ( pointers to 13 mnemonics )

: M" HERE SWAP 4 * MNEMS + ! 22 WORD C@ 1+ ALLOT ;
: ALIGN SP@ SP@ - ABS 1- DUP HERE + SWAP NOT AND DP ! ;

0 M" PSR " 1 M" VBR " 2 M" EPSR" 3 M" FPSR" 4 M" EPC " 5 M" FPC " 6 M" SSO "

```

```

7 M" SS1 " 8 M" SS2 " 9 M" SS3 " A M" SS4 " B M" GCR " C M" GSR "
ALIGN

( Dump all the control registers )
: .CREGS ( -- ) CR D 0 DO I 2 .R I CRN@ 9 U.R
  I 4 * MNEMS + @ COUNT SPACE TYPE CR LOOP ;

\ Vector table for interrupts
803C00 CONSTANT VECTORS ( out of the way on a 1K boundary

C50010 CONSTANT NIER ( normal interrupt enable register
C50045 CONSTANT PLSR5
C5005E CONSTANT PLSR30
C50000 CONSTANT ICR

( MCORE constants for registers )
CB0000 CONSTANT CR1
CB0001 CONSTANT CR2
CB0002 CONSTANT BR
CB0003 CONSTANT SR
CB0005 CONSTANT DR
CB0006 CONSTANT PURD
CB0007 CONSTANT PORTS
CB0008 CONSTANT DDRS
C5000C CONSTANT IPR

: SLAVE 0 DDRS C!
  C CR1 C! C CR2 C! SR C@ DR C@ 2DROP CC CR1 C! ;
: -SPI 0 DDRS C! 6 CR1 C! ;
: PORT -SPI 1 DDRS C! 0 PORTS C! ;
: .SPI ind C@ . samples 20 DUMP ;

: TEST 0 ind ! PORT SLAVE ;

: T TEST ;

: INIT SLAVE ;
: R DR C@ . ;
: W SR C@ DR C@ 2DROP DR C! BEGIN SR C@ UNTIL ;
: S SR C@ . ;
: P PORTS C@ 8 0 DO DUP 80 AND IF ." 1" ELSE ." 0" THEN 2* LOOP DROP ;
: Q PORTS C! P ;
: Z BEGIN BEGIN SR C@ ?TERMINAL OR UNTIL S R ?TERMINAL UNTIL ;
: X 0 DO I W LOOP ;

\ Take a run at it
1 CRN@ VECTORS 80 CMOVE \ grab current vectors
sir_spi 1 OR VECTORS 22 4 * + ! \ INSTALL SPI VECTOR WITH AF BIT SET
sir_pit 1 OR VECTORS 21 4 * + ! \ install PIT victor
VECTORS 1 CRN! \ SET VBR
2 PLSR5 C! \ LEVEL 2 FOR SPI
1 PLSR30 C! \ LEVEL 1 FOR PIT
6 NIER ! \ ENABLE INTERRUPTS
0 ICR H! \ VECTORED INTERRUPTS

```

```

80000140 0 CRN! \ ALLOW NORMAL INTERRUPTS
SLAVE          \ SET FOR SPI RECEPTION
S R           \ CLEAR UP ANY OLD DATA
\ PIT         \ START THE PIT BULL
\ PORT       \ TURN OFF SPI SET MISO LINE LOW
\ TEST       \ PUT THE MISO LINE HIGH AND THEN SWITCH ON SPI INTERRUPTS

```

If you download this file, it will install interrupts for PIT and SPI as well as create words in the Forth dictionary to access the C functions. The details are in the following C files and only will work for this version of MaxForth. The C code has been compiled with a port of the GNU C compiler for Mcore. The makefile is included.

### 8.6.3 Makefile

This makefile was used to invoke the GNU compiler and compile the files: pit.c spi.c makeword.c words.c.

```

NAME                = test io

S                   = ..
I                   = ../../Mcore
CC                  = gcc-mcore -I..\ -I..\Mcore\ -I..\..\Mcore
LD                  = gcc-mcore -nostartfiles -T gnuLink.lcf
AR                  = ar-MCORE
AS                  = as-MCORE
GS                  = gasp-MCORE
RM                  = del

.SUFFIXES : .o .c .s

CFLAGS = -c -O3
AFLAGS = -ahls

pit: pit.c gnuLink.lcf
spi.c makeword.c words.c      $(CC) $(CFLAGS) -Wa,-ahls -Wunknown-pragmas pit.c
oformat=srec,-Map=pit.map\    gcc-mcore -Ttext 0x806000 -nostartfiles -Wl,--
                              -o pit.s19 pit.o spi.o makeword.o words.o

gnuLink.acf:
forth.h:
micro.h:

```

### 8.6.4 gnuLink.acf

This file uses a linker language to describe what the target memory looks like and how the sections of code, data and empty space fit into it all.

```

MEMORY
{
  VECTORS (R) : ORIGIN = 0x00004000,          LENGTH = 0x00000004
  TEXT (RX) : ORIGIN = 0x00004004,          LENGTH = 0x00001000
  DATA (R) : ORIGIN = 0x00010000,          LENGTH = 0x00001000
  MAXFORTH(R) : ORIGIN = 0x0001FC00,          LENGTH = 0x00000400
  BSS (RW) : ORIGIN = 0x00800000,          LENGTH = 0x00001000
}

/* damn deadstrip prevention */
SECTIONS

```

```

{
beginvectormap :                               /* Put the vector map at the very
{
per */= ALIGN(0x400);                          /* Align table on 1024-byte boundary
__vector_table_start__ = .;                    /* MCore requirement */
*      (vectortable)                          /* File of vector table */
__vector_table_end__ = .;
} >VECTORS                                     /* Map to VECTORS memory section (0x000-0x1FF) */

.main_application :                             /* Application Code */
{
* (.text)
. = ALIGN(0x4);
* (.rodata)
. = ALIGN(0x4);
__sinit__ = .;
/* STATICINIT */
. = ALIGN(0x4);
__ROM_INIT_START = .;

} >TEXT                                         /* Map to TEXT section (0x200-) */

/* tell linker generate the Load address other than the real address */
.main_app_data : AT(__ROM_INIT_START)          /* Data Section */
{
. = ALIGN(0x4);                                /* A section of data to be copied to RAM */
__data_ROM_begin = .;                          /* Sets location variable used in rom_copy */
* (.data)
/* Just include data in all modules for now */

* (.vtables)
* (.exception)

. = ALIGN(0x4);
__exception_table_start__ = .;
/* EXCEPTION */
__exception_table_end__ = .;

__data_ROM_end = .;                            /* to be copied to RAM */
} >DATA

.maxforthmap :                                /* Put the vector map at the very beginning */
{
. = ALIGN(0x400);                              /* Align table on 1024-byte boundary per */
__maxforth_vector_start__ = .;
*      (forthvector)                          /* File of vector table */
__maxforth_vector_end__ = .;
} >MAXFORTH                                    /* Map to MAXFORTH memory section (0x1FC00) */

.main_app_bss :                                /* Uninitialized data space */
{
__bss_begin = .;
* (.bss)
__bss_end = .;

```

```

/* Calculations and assignments of section sizes      */
__stack_end   = .;
__stack_begin = __stack_end + 0x0800;
__stack_begin = (__stack_begin + 7) & ~7;

__heap_addr   = __stack_begin;           /* see MSL alloc.c */
__heap_end    = __heap_addr + 0x0000;
__heap_size   = __heap_end - __heap_addr;

__data_begin   = __data_ROM_begin; /* set to shadow RAM address */
__data_size    = __data_ROM_end - __data_ROM_begin;
__data_ROM_begin = __ROM_INIT_START;
__data_ROM_end   = __ROM_INIT_START + __data_size;
F_user_data_dict = __heap_end;        /* first memory location for user */

__exception_table_size = __data_ROM_end - __data_ROM_begin;
} >BSS
}

```

### 8.6.5 spi.c

```

// MCore end of SPI link to HC12  Rob Chapman  Nov 8, 02

// When the miso line is pulses low, then high, it is a signal to the HC12 that
// it should upload its current sensor readings.  To get the miso line low, the
// data direction register is set, the port is written to as a zero and the SPI
// is turned off.  Turning the SPI back on, will make the miso pin an input and
// it will go high, initiating values from the HC12 sensor queues.

#include "micro.h"
#include "forth.h"
#include "interrupt.h"

Byte samples[9*2+1],ind;

void sir_spi(void) // service the spi interrupt
{
    if (reg_SPIISR.reg);
    samples[ind] = reg_SPIDR;
    if (ind < (sizeof(samples) - 1))
        ind++;
    RTE; // return from exception
}

```

### 8.6.6 pit.c

```

// MCore PIT  Rob Chapman  Nov 17, 02

// A periodic timer is used to get sensor values from the HC12.  The interval
// is settable from 2 ms or more.

#include "micro.h"
#include "forth.h"

```

```

#include "interrupt.h"

extern ind;
Cell sample_no; // sample number

void sir_pit(void) // service the pit interrupt
{
    sample_no++; // increment sample number
    ind = 0; // reset index for sample array
    reg_SPICR1.reg = 6; // turn off SPI to gain port control
    reg_SPIDDR.reg = 1; // set the MISO line as an output
    reg_SPIPORT.reg = 0; // make sure its zero
    reg_SPIDDR.reg = 0; // turn it off now
    reg_SPICR1.reg = 0xCC; // turn spi back on
    reg_PCSR1.bit.PIF = 1; // turn off pit interrupt
    RTE; // return from exception
}

```

### 8.6.7 forth.h

```

/* header file for forth.c Rob Chapman Feb 2, 2000 */
/*
    17 mar 02 bjr - changed user_dict from Cell to Byte
    28 apr 02 bjr - removed UD_SIZE, dict now define in linker file
*/

// Since long long is not universally supported, the double cell support
// can come from either long for 16 bit cells, long long if supported
// for 32 bit cells; or forth double number support if no double support
// is available. For this reason, Duo and U_Duo are defined in micro.h

/* Portable type definitions */
typedef unsigned int Bits; // for bit fields */
typedef unsigned char Byte; // a byte */
typedef unsigned int Cell; // default memory unit; ok as pointer */
typedef Cell Flag; // a flag: 0 or non zero */
typedef signed int Integer; // for signed numbers */
typedef unsigned int Natural; // for unsigned numbers including 0 */
typedef void (*inner)(void); // inner interpreter

#define Forth void *

//typedef void (Forth)(void); // Forth word

#define Literal(A) (const Cell)(A)
#define Forward(A) Literal(A*sizeof(Forth))
#define Backward(A) Literal(-A*(signed)sizeof(Forth))
#define String(S) (S)

#define CONSTANT(S,N) Forth * const S[] = {(Forth)cii,(Forth)Literal(N)}
#define FUN_VECT (void (*)(void)) // good for casting vector addresses

// should be in micro.h, user dictionary size
// #define UD_SIZE 1000

```

```

/* flag */
#ifndef TRUE
#define TRUE ((Cell)-1)
#define FALSE 0
#endif

/* list Forth functions to be called */
extern inner *wp, **ip;
extern Cell *dsp, *rsp; /* data stack pointer; and return stack pointer */

extern void iboot(void);
//extern const Forth *tasknfa[];
extern const struct { unsigned char name[6]; const unsigned char *link;\
void **pfaptr;}tasknfa;

// should have a limit as well, maybe just documented or known.

//extern Forth * const task[];

/* virtual forth engine definitions */
typedef struct {
    void *inner;
    void (*code)(void); /* points to code like the inner interpreter */
} Quarkextra;

typedef inner Quark; /* fundamental building block */

typedef union param{
    void (*inner)(void); /* points to code like the inner interpreter */
    union param *body[1]; /* points to an array of pointers to these */
//    Forth whatever;
} Param;

#ifdef DUO
// Endiandependance for the double integer guys
// Forth model is top is upper cell and next is lower cell
// Little endian would have it switched from big endian (Forth)
// These little ditties run independant of endian and convert
// to duos and back from memory. Trick is, no conditionals.
Duo asduo(Cell *);
void toduo(Duo , Cell *);
#endif

```

### 8.6.8 micro.h

```

// M2107 Settings

#include "mmc2107.h"

#ifndef MICROH
#define MICROH

#define MCORE

```

```

// #define DUO 1
// typedef signed long long Duo;      /* double the integer power */
// typedef unsigned long long U_Duo; /* double the cell power */

// eeprom no here
#define EEPROM_START 0

// wrist canine chronometer
#define COP_OFF reg_WCR.bit.EN = 0      /* disable watchdog timer */

// external RAM
#define EXTERNAL_RAM

// assembler to forth call
#define ATO4 0xE198 // for hc12? or mcore?

extern unsigned int F_user_data_dict;

// user dictionary
#define SETUP_DICTIONARY *(Cell *)dppfa = (Cell)&F_user_data_dict;\
    /* value from linker file*\
    *(Cell *)fencepfa = (Cell)&F_user_data_dict; // value from linker file

// serial io
#define SEND(X) reg_SCI1DRL = X
#define RECEIVE reg_SCI1DRL
#define SENT reg_SCI1SR1.bit.TDRE
#define RECEIVED reg_SCI1SR1.bit.RDRF

// #define TDRE 0x80
// #define RDRF 0x20
#define FREQ 32 // frequency for 2114 chip
#define BAUD57 (FREQ * 1000000 / (16 * 57600))
#define BAUD96 (FREQ * 1000000 / (16 * 9600))
#define BAUD19 (FREQ * 1000000 / (16 * 19200))

#define SET_SERIAL COP_OFF;\
    reg_SYNCR.reg = 0x2000;\
    reg_SCI1BD = BAUD19; /* select the baud rate */\
    reg_SCI1CR1.reg = 0x00; /*reset values (8/N/1)*\
    reg_SCI1CR2.bit.TE = 1; /*enable transmitter*\
    reg_SCI1CR2.bit.RE = 1; /*enable reciever*\
    if (reg_SCI1SR1.reg); /*transmitter is actually enabled here*\
    while(!reg_SCI1SR1.bit.TDRE) /* wait for TDRE to go high */

#define _flash ((void **)0x190) /* vector for flash program module */
#define ALIGNED(a) (((Cell)a + 3) & ~3)

/* Stack sizes */
#define RS_SIZE 64
#define DS_SIZE 64
#define FS_SIZE 8

// User dictionary size

```



```

#define UD_SIZE 1000

// Environment
#define DICTSTART 0x800000
#define DICTEND 0x81FFFF
#define QUICKTAG 0x3FFC
#define QUICKVECTOR 0x3FF8
#define BOOTTAG 0x3FF0
#define BOOTVECTOR 0x3FF4
#define BOOTSTART 0x10000
#define BOOTEND 0x1FFFF

#endif

```

### 8.6.9 interrupt.h

```

#define RTE asm("rte")// exit interrupt

```

### 8.6.10 words.c

```

// MCore interrupt routine interface

#include "forth.h"
#include "makeword.h"

extern Byte ind;
extern Cell sample_no;
extern Byte samples[];
extern void sir_pit(void);
extern void sir_spi(void);

void hello(void)
{
    ind++;
}

void _start(void) // link these C words to the Forth dictionary
{
    FUNCTION(hello);
    VARIABLE(ind);
    VARIABLE(samples);
    VARIABLE(sample_no);
    VARIABLE(sir_spi);
    VARIABLE(sir_pit);
}

```

### 8.6.11 makewords.c

```

// Add to dictionary from C Rob Chapman Nov 20, 02

#include <string.h>
#include "forth.h"
#include "makeword.h"

#define list ((void **)0x800508) // points to last in dictionary
#define dp ((Byte **)0x80050C) // dictionary pointer

```

```

#define cii (inner *)0x4888          // constant inner interpreter

void make_header(char *s, void *m) // link word to dictionary
{
    Byte l = strlen(s);           // get length
    void **t = *list;             // get current nfa

    *list = (void **) *dp;        // set list to new word
    *(*dp)++ = 1|0x80;            // store count byte with msbit set
    while(l-- )                   // for the length of the string
        *(*dp)++ = *s++;          // create the name
    *(*dp)-1 |= 0x80;             // terminal byte
    while (((Cell)*dp & (sizeof(Cell)-1))) // alignment
        *(*dp)++ = 0;            // zero and increment
    *((void ***)dp)++ = t;        // add link to current word
    ***(void ***)dp = (void **)(*dp+2*sizeof(Cell)); // pfa pointer
    *dp += sizeof(Cell);         // next cell
    ***(void ***)dp++ = m;       // point to C routine
}

void make_variable(char *s, void *m) // link as variable constant
{
    make_header(s,cii);           // make it as a constant
    ***(void ***)dp++ = m;       // point to C variable
}

makewords.h

#define VARIABLE(A) make_variable(#A,&A)
#define FUNCTION(A) make_header(#A,&A)

void make_header(Byte *, void *);
void make_variable(Byte *, void *);

```