MPE IRTC Forth for the ATMEL AVR series

User Manual

MPE IRTC Forth for the ATMEL AVR series

User Manual

Manual revision 1.100

Date 22 September 1999

Software

Software version 1.100

Package Number:	

For technical support

Please contact your supplier

For further information

MicroProcessor Engineering Limited
133 Hill Lane
Southampton SO15 5AF
UK
Tel: +44 (0)2380 631441
Fax: +44 (0)2380 339691

e-mail: mpe@mpeltd.demon.co.uk
tech-support@mpeltd.demon.co.uk
web: www.mpeltd.demon.co.uk

Acknowledgements

The source code for this product was licensed from:

RAM Technology Limited

It has been modied and enhanced for use on MPE's ProForth VFX for Windows which is the host Forth, and for use with MPE's AIDE development environment.

Licence terms

Notice

RAM Technology Systems and MicroProcessor Engineering MAKE NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. RAM Technology Systems shall not be liable for errors contained herein or for incidental consequential damages in connection with the furnishing, performance, or use of this material.

The information contained in this document is subject to change without notice.

Distribution of application programs

Binary code produced with this compiler and tools may be freely distributed without royalty. No source code provided by MPE or RAM Technology may be distributed without written permission first being granted.

Warranties and support

We try to make our products as reliable and bug free as we possibly can. We support our products. If you find a bug in this product and its associated programs we will do our best to fix it. Please check first by fax or email to see if the problem has already been fixed. Please send us enough information including source code on disc or by email to us, so that we can replicate the problem and then fix it. Please also let us know the serial number of your system and its version number. We will then send you an update when we have fixed the problem. The level of technical support that we can offer may depend on the Support Policy bought with the product.

Technical support will only be available on the current version of the product.

Make as many copies as you need for backup and security. The issue discs or CD are not copy protected. The code is copyrighted material and only ONE copy of it should be use at any one time. Contact MPE or your vendor for details of multiple copy terms and site licensing.

As this copy is sold direct and through dealers and purchasing departments, we cannot keep track of all our users. If you fill out the registration form enclosed and send it back to us, we will put you on our mailing list. This way we will be able to keep you informed of updates and new extensions, as they become available. If you need technical support from us we will need these details in order to respond to you. You will find the serial number of the system on the original issue discs.

Contents

1 Introduction

Introduction

Welcome to the world of the Interactive Remote Target Compiler (IRTC) and the Target Link Monitor (TLM) for the ATMEL AVR series of processors. This package is optimised for use with the RAM Technology In-System programmer (ISP), which is supplied as part of the package.

These products are designed to give fast and efficient turn-round of ATMEL AVR based microcomputer products written in ANS Forth. IRTC is simple to use but very powerful. It is fast in compilation and allows good product documentation from the keyboard without the need for large manuals.

For those not familiar with Forth we would advise you start with one of the books in the Appendix.

Via the TLM and ISP, IRTC provides a seamless integration of the Host PC and the Target hardware. This provides for the interactive testing of programs and hardware from the PC keyboard. Test routines may be compiled and immediately run in the Target. Memory may be viewed, variables modified, and the source code or help of any compiled word displayed.

As a consequence of the low cost of the IRTC and TLM we regret that is not possible to offer a voice Technical Hot Line. We are, however, able to offer e-mail for your enquires. These will be dealt with as soon as possible.

Our E-mail is: tech-support@mpeltd.demon.co.uk

Web Page http://www.mpeltd.demon.co.uk

2 Installation and Configuration

System Requirements

IBM-PC with Windows95/98 or NT

- 16Mb of RAM
- COM1: Serial Port
- Hard Disk
- Mouse
- ISP Programmer (supplied)
- Target system
- +5V power source of at least 50 mA capability to power the ISP serial programmer.

The ISP must be connected to a serial port before running IRTC.

Hard Disk Installation

Place the IRTC CD in the CD drive and run the Setup. The setup wizard will install the software on to your hard drive in the directory of your choice. The default is **`C:\XAVR`**.

The compiler and tools are run from the MPE development shell, **`AIDE`**. Make a short cut to **`AIDE`** on your desktop.

Configuration

Run AIDE from the file **`AIDE.EXE`** in **`XAVR\AIDE`**. There is a separate manual for AIDE.

In `IDE -> MacroManagement` set the AVRBASE macro to `C:\XAVR` or the directory where you have put the system.

In `IDE -> MacroManagement` set the ACROBAT macro to the full path name of the PDF file viewer (normally Adobe Acrobat), for example:
`C:\APPS\ACROBAT\ACRORD32.EXE`.

Edit the macros in the file **`XAVR\DEMO\DEMO.CTL`** to reflect the directories you are actually using.

Test that the system is working by clicking on the **AVR** icon on the top left toolbar.

The ISP is connected to your IBM® compatible computers serial port, COM1 to COM4. IRTC sets the port configuration to 38400 baud 8 bits one-stop-bit and no-parity. The ISP 25 way D-type connector is then plugged into your computer port or via a cable.

If the ISP is not connected to COM1, edit the **`KERNEL\*.INI`** files to use SET-COMx where xis 1..4.

The header on the 10 way strip cable is plugged into a pin header on your Target hardware. The Target powers the ISP through the strip cable via the power pins from your Target. See the ISP description.

Now read the rest of the manual, or go to the DEMO chapter.

3 Overview

The philosophy behind IRTC is that of a high level interactive de-bugger. Through TLM the AVR is made into a stack machine with three functions.

- Data may be transferred from the Host-PC to the TLM stack.
- Data may be transferred from the TLM stack to the Host-PC.
- The TLM may execute code at an address placed on its stack.

These three functions are all that is required to test individual Forth words or assembly subroutines. Then words or subroutines are added to the TLM code to extend it into an application. With the C@, file fetch, and C!, file store, routines added the TLM may test the hardware external to the AVR, as we then have the capability to change any file space location, ports and registers included. The TLM may be seen as a server to the Host and programs may be written to exercise the Target hardware without any further AVR code being compiled. This ability to test the hardware, before any application specific software is written, is one of the main advantages of IRTC. Being Forth based, this process is interactive with direct hardware operation from the command line.

XAVR.EXE is the base compiler that incorporates IRTC and AVR generic facilities. To produce an an application, load a CPU specific INI file that defines processor specific features, and then load the application code. This is normally done by loading a file we call a **control** file, usually with a CTL extension.

You can run XAVR.EXE from the command line or a short cut:

```
XAVR.EXE  GET DEMO.CTL
```

However, we recommend running it from the AIDE environment which gives additional useful features.

Selecting the right AVR CPU

For AVR Target configuration in AVR xxxx.INI, see Target Configuration.

If your application is not going to use the Mega103, change the **XAVR\DEMO\DEMO.CTL** file to select the correct INI file from the XAVR\KERNEL directory. IRTC is modified by the .INI file to reflect the properties of the AVR device chosen.

Once loaded it is NOT possible to change the AVR type selection without restarting IRTC with another .INI file.

New AVR xxxx.INI files may be produced to specify new devices as they become available. Existing .INI files may be used as a template to produce your own.

Loading your AVR with a TLM

IRTC runs the ISP in two modes, Programming and Interactive Communication. The serial programming mode is as described in the Atmel databook, the interactive communications uses the same pins, MISO, MOSI and serial communication at the ISP baud rate, 38400 baud. For the interactive mode to operate a small program, the Target Link Monitor, must first be programmed into the AVR Flash code space. An image of this is created by xxx.INI at load time and this must first be programmed into your device before development may begin.

You may program the TLM with **REPROGRAM**. The ISP and Target must be powered and connected to a selected serial port. No external connection , except the ISP, should be made to the Reset pin on the Target device or to the MISO, MOSI and SCK pins. **REPROGRAM** erases

the AVR Flash before programming the code image from $0000 to **HERE** (the end of the image).

When successful the AVR will now be ready for development.

Host and Remote modes

When the compiler is running, it is in one of two modes, Host or Remote.

Host mode is used when you wish to use the PC and its resources without any communication with the Remote Target. You may wish to perform some inspection of the Target image memory for example. All references to Forth kernel words, like @, !, + etc., will be from the Host definitions. That is those definitions that relate to the 80X86 processor in the PC.

After you type **?REM**, Remote mode will be announced. This indicates that where applicable IRTC will use the Remote Target for execution of the Target word definitions. This means that you now have two Forth systems running together. The Host controls everything through IRTC but execution occurs in the Remote Target. When you type on the PC keyboard the Host parses your input and performs the necessary actions to instruct the Remote Target what to do. The Target is a slave, but does execute the run-time code in real time.

This bit is for Forth techies, and don't worry if you don't understand it yet. The Vocabularies that will be searched are modified by the mode selection. When in the Host mode the first two will be **METAAVR**. This is the IRTC main vocabulary. When in the Remote mode the first two will be **METAAVR** and **TARGET**. The **TARGET** vocabulary holds all the AVR Forth definitions for the Remote Target. This will be the first vocabulary searched, so **@**, **!**, **+** etc. will now be the Remote versions in the AVR.

The following Forth words are useful for switching modes.

?REM switches to REMOTE mode if not already there

HOST switches to Host mode.

REPROGRAM programs the target AVR with the code compiled so far. This must be done before switching to Remote mode. Once in Remote mode, the compiler will automatically update the target memory.

VERIFY checks the target Flash image against the compiler's image.

.STATISTICS tells you useful things such as how much of your code space has been used up.

Running the DEMO application

Run AIDE.

Configure the macros and IDE as described earlier.

Click the AVR icon. The compiler will run, and when it finishes type

```
REPROGRAM      \ sets up the Flash
?REM           \ switch to remote mode
GO             \ the main application word
               \ ignore the NO Ack! Message
TTY            \ Start the simple terminal emulator
               \ press ESC to finish
```

What you should see is:

```
ABCDEFGHIJ 123456789 ABCDEFGHIJ
ABCDEFGHIJ 123456789 ABCDEFGHIJ
```

```
ABCDEFGHIJ 123456789 ABCDEFGHIJ
ABCDEFGHIJ 123456789 ABCDEFGHIJ
…

Done
```

The screen should clear and the above appears and runs until you stop it by pressing the <ESC> key. The result is not too impressive but what is happening is quite interesting. TTY is just a Dumb Terminal. It accepts serial input and puts it on the screen. There are three tasks, described later, that result in these characters appearing.

The first is called LETTERS, sends the string 'ABCDEFGHIJ' to a 'pipe' (See Pipes). The second NUMBERS, sends the converted number string '123456789' CRLF to the same pipe. The third UART, takes one character at a time out of the pipe and sends it back to the Host. The rate at which the characters come back is controlled by a Timer interrupt running every 111mSecs.

So you see the Target is quite busy. The tasks LETTERS and NUMBERS are endless, they just run and run. UART only runs once and then STOPs. The Timer interrupt AWAKENs UART and it runs once again. So the characters occur at a 111mSec rate.

The funny thing is why do the letters and numbers, that are being put in the same pipe, not get jumbled up and why two sets of letters and only one of numbers with the CRLF after letters and not numbers?

The answers to these and other questions come later. (see Semaphores)

Saving a target image

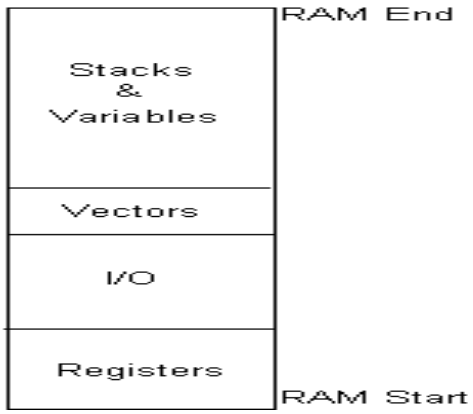
You can save the target image in image (straight binary) or Intel Hex format using:

```
SAVEIMG <filename>
SAVEHEX <filename>
```

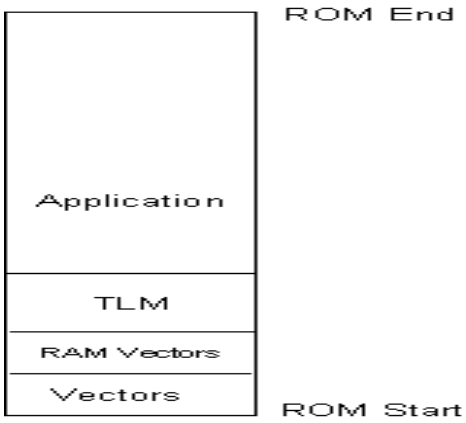
Note that Intel Hex files can be loaded into the standard **AVRDEBUG.EXE** program for disassembly and simulation.

4 Memory and AVR Registers

RAM



ROM



Separate Code and Data

The AVR has separate code and data spaces.

The following words operate on DATA space.

```
C@ C! @ ! 2@ 2! FILL ERASE BLANK CMOVE ACQUIRE RELINQUISH
```

But the code memory space uses:

```
MC@ M@ ,-T C,-T.
```

AVR Working Registers

AVR Reg name	AVR Reg number(s)	Forth use	Other Function
Z, ZH, ZL	R31:30	Scratch	Used for IJMP
Y, YH, YL	R29:28	Data Stack Pointer	
X, XH, XL	R27:26	TOS (top of data stack)	

	R25:24	SEC (intermediate)	
	R23:22	UP (user pointer)	
	R21:20	N+5:N+4 (scratch)	
	R19:18	N+3:N+2 (scratch)	
	R17:16	N+1:N (scratch)	

5 Target Configuration files

The way the Target AVR is used is defined bye the file **xxxx.INI**.

This is loaded by IRTC and extends XAVR.EXE by compiling the AVR set-up words and the talker, TLM, code. The positions and size of the memory spaces and stack positions are set.

Below is the source for **MEGA103.INI**;

```
\ Initialisation File for MegaAVR

38400 8 NOPARITY ONESTOPBIT SET-COM1
META IN-META
SET-FF
IN-META
$0060 VALUE RAM-VEC
$00A0 VALUE RAM-START
$0FFF VALUE RAM-END
$0000 VALUE ROM-START
$FFFF VALUE ROM-END
$0000 VALUE EPROM-START
$0FFF VALUE EPROM-END

\ Vectors

$0000 CONSTANT RESET
$0002 CONSTANT EXT-INT0
$0004 CONSTANT EXT-INT1
$0006 CONSTANT EXT-INT2
$0008 CONSTANT EXT-INT3
$000A CONSTANT EXT-INT4
$000C CONSTANT EXT-INT5
$000E CONSTANT EXT-INT6
$0010 CONSTANT EXT-INT7
$0012 CONSTANT TIM2-COMP
$0014 CONSTANT TIM2-OVF
$0016 CONSTANT TIM1-CAPT
$0018 CONSTANT TIM1-COMPA
$001A CONSTANT TIM1-COMPB
$001C CONSTANT TIM1-OVF
$001E CONSTANT TIM0-COMP
$0020 CONSTANT TIM0-OVF
$0022 CONSTANT SPI-STC
$0024 CONSTANT UART-RXC
$0026 CONSTANT UART-DRE
$0028 CONSTANT UART-TXC
$002A CONSTANT ADC-INT
$002C CONSTANT EE-RDY
$002E CONSTANT ANA-COMP

multi-tasking [if]
  GET MTMEGA         \ Multi tasking taker
[else]
  GET STMEGA         \ Single tasking taker
[then]

FORTH ' (PAGED-DOWNLOAD) IS (DOWNLOAD)

H: "MEGA S" Mega103 " ;
```

```
FORTH ' "MEGA IS AVR-TYPE

IN-META

HOST

.STATISTICS

CR CR .( IRTC for Mega103 Running in HOST mode )
```

The start and end of the file space, code space and EEPROM space are set. These are used to warn if exceeded during compilation. The programming algorithm is set for a paged Flash device and the TLM object code is loaded into the PC Target image.

The baud rate for the TLM may be modified in the talker startup code, but this must be done along with a similar modification to the ISP code if the baud rate value is to be changed

6 Optimisation

By default, the IRTC compiler creates subroutine threaded code with optimisation. The following directives can be used to control the optimiser:

```
FAST-CODE     \ enable optimiser
COMPACT-CODE  \ disable optimisier
```

As the compiler generates machine code it is possible for it to scan the code as it is compiled and to optimise the result. This process operates in a single pass, so the optimisation is limited, but is very useful.

Forth requires parameters on the stack before executing the function. If these parameters are literals it is often possible to reduce the code e.g..

```
 : TST $03 PORTA C! ;
```

This would require the following code;

```
PUSHT                \ Make room
LDI TOSL,03          \ literal 03
LDI TOSH,0
PUSHT                \ Make room
LDI TOSL,PORTA       \ literal PORTA
LDI TOSH,0
CALL C!              \ CALL C! routine
RET                  \ finished
```

However, the optimizer produces:

```
LDI SECL,03          \ literal 03
STS PORTA,SECL       \ Store to portA
RET                  \ finished
```

This has saved 6 words of storage and several micro seconds of execution time. It would be unusual to make this simple code a definition and so this would normally become in-line code within a larger definition. Also in this example, if the literal 03 was computed at run-time the result is already in TOS so the compiler just saves TOSL into PORTA and then cleans the stack.

7 Quick Forth Guide

First words

Good organisation and effective communication require us to:

- define and name useful tasks or operations
- group these into larger related ones.

This is how Forth works. It consists of a dictionary of **words** grouped into **vocabularies**. Each word is an operation and words are defined in terms of those that went before. This dictionary defines Forth itself and is added to by your application to grow into a new Forth that solves your problem.

Let us look at an example of how this works for a simple FAX machine.

```
: send ring read transmit hang-up ;
```

The **`:`** in Forth indicates the start of a new definition. The space delimited text following will be the name of the next entry in the dictionary. These names are called "words" in Forth. The other words already exist in the dictionary and are the functions performed by this new definition. The **`;`** terminates the new dictionary addition. This process is called compilation and the finding of the words in the dictionary interpretation.

The words **`ring`**, **`read`**, **`transmit`** and **`hang-up`** are defined in terms of other words, for example the word **`read`**:

```
: Read BEGIN scan-line page-end UNTIL ;
```

Here we see read defined in terms of other words, this time with a looping structure that terminates at the page end. Notice that the definitions are short and the word names relevant. Also we have shown the definitions in a top-down style, which is how you might consider the problem, but they would be implemented in a bottom-up fashion. This means that the simplest task is defined and tested first, building up to the final function send.

The words in **`read`** may well consist of what are called Forth primitives as are the **`BEGIN...UNTIL`** structure in **`read`** itself. These primitives are the fundamental functions of Forth and are the starting point from where the Forth grows or extends. Hence the Forth language is termed "extensible".

To recap, Forth is an extensible language that consists of a dictionary of words each defined in terms of other words.

Interactive

One of the many advantages of Forth is it's ability to execute a word from the dictionary by name. Once a colon definition has been entered it may be run, or executed, just be typing the word name and pressing the RETURN key.

Forth is thus called an interactive language because it carries out your commands as soon as you enter them at the keyboard.

When the RETURN key is pressed Forth tries to run all the words on the command line and at the end prints:

```
ok
```

This is true providing no problem was found. For instance if a word is not in the dictionary Forth will say:

```
Error: <word> is undefined
```

Or one of the words may not have enough parameters and you may see:

```
stack underflow
```

The Stack

Forth is a stack based language. This means that all the parameters used during operations or calculations are held on a stack. The stack is a Last-in-First-Out, LIFO, structure in the re-writable area of computer memory. If we put a value on the stack we say we are 'pushing' it onto the stack. If we retrieve it we say we are 'popping' it off the stack.

This stack operation is very useful but leads to some differences. Suppose we wish to add two numbers:

```
PRINT 1 + 2 3
```

This is how we might look at the problem in Basic. In Forth we would write:

```
1 2 + . 3 ok
```

Here the values are put first and the operator last. This is called 'post-fix' or Reverse Polish Notation (RPN), we normally use 'pre-fix'. The . word outputs the top stack item converted to an ASCII number string to the PC screen.

The advantage to the computer is that it can only handle postfix order as it does everything sequentially. It must have the values before it can do the addition. In Forth the entry of the two numbers pushes them onto the stack. The 2 was last so it is on the top of the stack with the 1 underneath it. When Forth executes the + code it only knows that it must add the top two stack values together and leave the answer on top of the stack. The + code does not need a location for the two numbers it just uses whatever is on the stack. If we performed:

```
1 2 + 5 - . -2 ok
```

The answer of -2 would be left on top of the stack. No intermediate storage of the 1+2 is required. This requires less memory and is dynamic with no need to 'garbage-collect' as there is no 'garbage'.

Stack Nomenclature and Documentation

All Forth functions operate with the stack for parameter passing. Any number of stack values may be input and output from a definition so we show the stack use of a word when we define it. To do this **(--)** and **(S --)** are used as stack comments.

```
(S n1 n2 -- n3 )
( n1 n2 – n3 )
```

Would mean that two values were required on entry and one value is left on exit from the word. Operands of differing sizes and types are shown below.

```
n1 n2          16 bit signed numbers
d1 d2          32 bit signed numbers
u1 u2          16 bit unsigned numbers
a1 a2          16 bit addresses
fa1 fa2        8 bit file addresses
b1 b2          8 bit bytes
```

```
c1 c2          ASCII characters
f              8 bit flag; 0 = false, 255 = true
```

AVR Forth

After the general discussion of Forth we may now look at the Forth implementation used for the AVR. IRTC creates subroutine threaded code with optimisation if the **FAST-CODE** directive is used. Many Forth words are compiled as in-line code to further improve execution speed.

Data Stack Size

The Forth Data stack size is 16 bits. Eight bit byte values are represented as a 16 bit value with the upper 8 bits set to zero. 32 bit, double, values are represented by two stack values, the top value the most significant. The top stack value is always in the **TOSL**, **TOSH** registers, R26 and 27, the X register. The Y register, R28, **DPL** and R29, **DPH**, holds the pointer to the rest of the data stack. Macros **PUSHT** and **POPT** push and pop the top stack item on and off the external stack. Also R24 and R25 are used as a temporary second stack item, **SECL** and **SECH**, that has macros **PUSHS** and **POPS**.

Return Stack

The return stack for the Forth word calls is controlled by the stack pointer **SPL** and **SPH** in the AVR I/O space. The **PUSH** and **POP** mnemonics control the stack along with **CALL** and **RET**.

Data Stack Operators

DUP SWAP OVER ROT DROP NIP TUCK

As all the parameters are passed on the stack Forth has several words to manipulate the stack contents.

DUP (S s1 - s1 s1) will duplicate the top stack item, leaving two identical values.

SWAP (S s1 s2 - s2 s1) exchanges the top two stack items.

OVER (S s1 s2 - s1 s2 s1) copies the second stack item onto the top of the stack, pushing down the first.

ROT (S s1 s2 s3 - s2 s3 s1) brings the third stack item to the top of the stack and the top and second are pushed down.

DROP (S s1 -) removes the top stack item, opposite to duplicate.

NIP (S s1 s2 - s2) removes the second stack item.

TUCK (S s1 s2 - s2 s1 s2) copies the top stack item beneath the second stack item.

These operators allow us to use the stack for local storage during words and re-ordering the results to pass on to the next word.

Memory Operators

C@ C! @ ! MC@ M@

The AVR has two memory areas to which it has access, code and RAM. **C@** and **C!** fetch from and store to a single 8 bit RAM location. These two words are optimised during compilation if fixed values, literals, are used as the operands. By using I/O before an I/O register **C@** and **C!** may be used to access the I/O locations as though they are RAM. The words **@** and **!** are 16 bit RAM operators. All these use a 16 bit address to access the location.

The code space in the AVR is only accessible indirectly via the Z register and the LPM instruction. Tables may be made using **CREATE** to construct lists of values in the code space with **C,-T** and **,-T**, which are words the compiler uses to store bytes and words into the code space. When the table name is later executed it leaves a 16 bit address on top of the stack that points to the beginning of the list. **MC@** and **M@** may then use this address to return the stored values.

Note: The address used by **MC@** and **M@** is a byte address not a word address. **CREATE** leaves a word address and this must be doubled.

```
CREATE 7-SEG.DIGITS (S - a )
( 0 ) $3F C,-T ( 1 ) $06 C,-T ( 2 ) $5B C,-T ( 3 ) $4F C,-T
( 4 ) $66 C,-T ( 5 ) $6D C,-T ( 6 ) $7D C,-T ( 7 ) $07 C,-T
( 8 ) $7F C,-T ( 9 ) $6F C,-T

 : GET-DIGIT (S b - b' ) 7-SEG.DIGITS 2* + MC@ ;
```

Here the table returns an address which is doubled by **2*** and added to the offset of the digit required by **+**. **MC@** uses the new byte address to fetch the value to the stack.

Logical Operators, AND OR XOR NOT

In embedded controllers the need for logical operators is paramount for the bit control necessary.

AND (S s1 s2 - s3) creates the logical AND of the top two stack items.

OR (S s1 s2 - s3) creates the logical OR of the top two stack items.

XOR (S s1 s2 - s3) creates the logical XOR of the top two stack items.

INVERT (S s1 - s1) inverts all the bits of the top stack item.

The code produced is in-line and optimised.

Bit Control, ON OFF CHECK

These are **TRANSITION**al definitions that do not appear in the Library. Their Library counterparts are **SET-BITS**, **RES-BITS** and **TEST-BITS**.

ON (S m fa -) takes an 8 bit mask and address and sets all the bits set in the mask high at the address leaving the rest unchanged.

OFF (S m fa -) takes an 8 bit mask and address and clears all the bits set in the mask low at the address leaving the rest unchanged.

CHECK (S m fa - m') returns the logical AND of the 8 bit mask and the contents of the address.

These are only 8 bit functions and operate on any file address. They may be used to manipulate flags, registers or ports producing tight code.

Maths, + - UM* UM/MOD

+ (S s1 s2 - s3) adds the top two stack items leaving the result on top of the stack.

- (S s1 s2 - s3) subtracts the top stack item from the second leaving the result on top of the stack.

UM* (S s1 s2 - s3 s4) multiplies the top two stack items to leave a double result on top of the stack i.e. 8*8=16, 16*16=32 bits.

UM/MOD (S s1 s2 s3 - s4 s5) divides the double value under the top stack item by the top stack item leaving two values, remainder second and quotient on top i.e. 16/8=8+8. 32/16=16+16 bits.

Increment and Decrement, 1+ 2+ 1- 2- 2* 2/

1+ (S s1 - s2) increments the top stack item by one.

2+ (S s1 - s2) increments the top stack item by two.

1- (S s1 - s2) decrements the top stack item by one.

2- (S s1 - s2) decrements the top stack item by two.

2* (S s1 - s2) arithmetic left shift of the top stack item by one.

2/ (S s1 - s2) arithmetic right shift of the top stack item by one.

Constants and Variables

Often we require a value that is fixed and used throughout our application, for example the bit on a port. This may be compiled as a **CONSTANT**;

```
2 CONSTANT PUMP
```

IRTC will cause this to be compiled as an in-line literal whenever the word **PUMP** is encountered. If the **COMPACT-CODE** directive is set, a subroutine will be compiled immediately and all references will be via calls. This may save code space but does not allow the optimiser to reduce code.

When the value required needs to be changed during execution a **VARIABLE** is defined;

```
VARIABLE COUNTER
```

Here **VARIABLE** compiles an in-line literal of the address of a 16 bit, two byte, location in the file. These locations start at the address in **RAM-START** and an error is generated if too many variables are declared resulting in the pointer going above the value **in RAM-END**. The variable allocation pointer is incremented automatically by each variable declaration. The pointer may be set absolutely by **VORG**. **VARIABLE** allots two bytes and **CVARIABLE** one byte. **@** and **C@** may be used to access the contents, **!** and **C!** to change the value.

Note: The contents of a variable location are undefined at compile time. It is necessary to initialise the contents at the start of your application if not implicitly done so by your code.

Control Structures

The flow of control in a program is normally sequential; control structures allow you to alter this by a branch or loop. The change is often brought about by a piece of (runtime) data. This gives the program the ability to choose which direction it will take.

Control structures must be used inside a : definition and may not start in one definition and finish in another. They cannot be used directly from the command line. But they may be nested inside one another as long as they do not overlap.

IF ... THEN

Use: **flag IF true words THEN**

The 16 bit flag controls the outcome and is consumed by IF. If the flag is non-zero the true words are executed, otherwise they are not.

If the flag is required again it may be duplicated by **DUP**. If the flag value is only required between **IF .. THEN** it may be conditionally duplicated by **?DUP**.

IF ... ELSE ... THEN

Use: **flag IF true words ELSE false words THEN**

This is similar to **IF ... THEN** above but with execution phrases for both true and false (zero) values of the flag.

BEGIN ... AGAIN

Use: **BEGIN words AGAIN**

This structure forms an endless loop that is only terminated by a AVR reset. This is often the structure used in your application run word.

BEGIN ... UNTIL

Use: **BEGIN words flag UNTIL**

This structure forms a loop that is always executed at least once. The loop will return to **BEGIN** as long as the 16 bit flag is false (zero) when tested by **UNTIL**. As soon as flag is true (non-zero) the loop is terminated. The flag is consumed by **UNTIL** in both cases.

BEGIN ... WHILE ... REPEAT

Use: **BEGIN words flag WHILE words REPEAT**

This is perhaps the most powerful of the Forth control structures. The loop starts at **BEGIN** and executes all the words as far as **WHILE**. If the 16 bit flag on the data stack is true (non-zero) the words between **WHILE** and **REPEAT** are executed, and the cycle returns to **BEGIN**. If the flag tested by **WHILE** is false (zero) the loop is terminated. **WHILE** consumes the flag.

CASE ... OF ... ENDOF ... ENDCASE

Use:

```
CASE (S b -- )
  value1 OF  words  ENDOF
  value2 OF  words  ENDOF
  ...
  default words ( otherwise clause )
ENDCASE
```

The case statement used in IRTC is the result of a competition run by the Forth Interest Group (FIG) in the USA. It was invented by Dr. Charles E. Eaker, and was first published in Forth Dimensions, Vol. II No. 3.

CASE exists to replace large and unwieldy chains of **IF ... ELSE ... THEN** statements.

The function of **CASE** is to execute one action dependent on the 16 bit value passed to it on the stack. The **OF** words compare value1,2 etc. to the stack value and if equal the words between **OF** and **ENDOF** are executed, the structure is then exited. If no match is found the

words between the last **ENDOF** and **ENDCASE** are executed with the input stack value still on the data stack.

FOR ... NEXT

Use: **index FOR words NEXT**

This is the simplest counted loop in IRTC. The index is transferred to the Return Stack and the words between **FOR** and **NEXT** executed. **NEXT** decrements the index and if true (non-zero) returns to **FOR**. If the index decrements to zero, false, it is removed from the Return Stack and the loop terminates.

DO ... LOOP

Use: **limit index DO words LOOP**

DO transfers the 16 bit values of the loop limit and the start index to the Return Stack. The words between **DO** and **LOOP** are executed at least once. **LOOP** increments the index and tests it against the limit. If the index is still less than the limit the loop executes again. If not the Return Stack is cleared and the loop exits.

To leave the loop early use **LEAVE**. This forces the index to be the same as the limit so that the loop will exit when **LOOP** is next run.

Conditional Tests

0= (S s1 - f) the flag is true if the top stack item is zero.

0< (S s1 - f) the flag is true if the top stack item is less than zero.

0> (S s1 - f) the flag is true if the top stack item is positive, greater than zero.

= (S s1 s2 - f) the flag is true if the top two stack items are equal.

<> (S s1 s2 - f) the flag is true if the top two stack items are not equal.

< (S s1 s2 - f) the flag is true if s1 is less than s2.

> (S s1 s2 - f) the flag is true if s1 is greater than s2.

U< (S s1 s2 - f) the flag is true if s1 is logically less than s2.

U> (S s1 s2 - f) the flag is true if s1 is logically greater than s2.

Seeing is believing

As IRTC creates AVR machine code and optimises it as it goes it may be reassuring to see the result. At the moment there is no de-compiler in IRTC. However, it is possible to create a HEX file of your code with;

```
0 HERE-T HEX-SAVE TEST
```

We may view the code by using the Atmel Studio debugger and step through the code with it.

Controlling a Port

Now we have communication we may modify any of the RAM locations, hence the ports. The command **FDUMP** will list all the file locations and **C@** will return the value of a single location.

To read from PortA all we need is:

```
I/O PORTA C@ H.
```

This returns the value of PortA.

all the registers and flags are known to IRTC, if you use one from a device other than the one chosen an error will be issued.

To write to PortA, first DDRA must be set to an output on the port pin we wish to use. Say we are using bit 0, then we could set this to an output with:

```
$01 I/O DDRA C!
```

The port bit may then be set high with:

```
1 I/O PORTA C!
```

or low with:

```
0 I/O PORTA C!
```

The same applies to any other port timer or internal function.

Comments

Comments may be incorporated into your source in three ways:

`COMMENT: … COMMENT;` or **`(( … ))`**

`\ to end of line`

`( … )` or **`(S … )`**

`COMMENT:` to **`COMMENT;`** is used to enclose several lines of comment.

The **`\`** ignores any further text on that line only. Starting a block of lines with **`\`** may be used instead of **`COMMENT:.`**

The **`(`** or **`(S`** ignore all text until the **`)`** is encountered. This is used for the stack comments.

Displaying Comments

if you wish to print out a remark or comment while a file is being interpreted or compiled use:

```
CR .( Comment string )
```

This will issue a Return and then print out the string on the Host terminal.

Loading a File

Once a file has been edited it may be compiled to produce the AVR runtime code with:

```
FLOAD <filename>
INCLUDE <filename>
GET <filename>
```

If the file has a .FTH extension, the default, no extension need be added to <filename>. **`FLOAD`** scans the file from beginning to end and interprets the contents from the dictionary. If you are in the Host mode the Target image only is updated. If in Remote the code is also programmed into the AVR for immediate execution.

INCLUDE and **GET** are aliases for **FLOAD** and may be used instead if you prefer. This reads better when used to load, include, a file from within a file.

Vocabularies

The Forth dictionary is divided up into sections called vocabularies. Each contains words that are related and this subdivision means that only a part of the dictionary need be searched to find a function.

The Forth word **ORDER** shows a list of vocabularies that represent the current search order. The one at the top of the list, highlighted, is called the **CURRENT** vocabulary and is the one that new definitions will be entered into. The list below starts with the **CONTEXT** vocabulary which is the first one to be searched to find a word being parsed. If the word is not found the next vocabulary in the list will be searched, and so on until the word is found or the list expires. IRTCis set to remove the necessity for you to need to modify this order during normal operation. Only is you wish to modify the compiler or do special operations to create new Library definitions will this order need to be changed.

You will notice that the Host and Remote modes have different vocabularies selected. Below is a list of the functions and their vocabularies.

ROOT
This is the root vocabulary and is always present. It contains words that allow the search order to be changed and the other vocabularies to be selected.

FORTH
This is the main vocabulary that contains the usual Forth words. These are the Host definitions that run IRTC and relate to the PC processor.

METAAVR
Here are the meta-compiler functions that make up IRTC.

XASM
The AVR assembler mnemonics, registers and flags are in here. This is only searched if a **CODE** or **C[…]C** definition is being compiled.

TRANSITION
Many of the words in here are replicas of those in **FORTH**. These are the optimisers for IRTC and this vocabulary is the first to be searched during a Target **:** definition.

LIBRARY
These are the Library macros that compile the Target code and create the Target vocabulary entries.

TARGET
The Target vocabulary contains the words that exist in the AVR code space. These words may be interactively executed. All Target : definition words go here together with any Library definitions used during compilation.

When changing from Host to Remote or back the vocabulary order is also changed. This order is re-established whenever the <cr> key is pressed. If you change the order it will only last for one command line of input.

When entering **CODE** definitions from the command line, each new line should start with **XASM**. **:** definitions may only be one line in length.

A list of words available in each vocabulary may be seen with:

```
<vocabulary-name> WORDS
```

CREATE ... DOES>

This construct is what makes Forth somewhat unique. It allows the compiler to be extended. In fact the assembler is written using this construct to define words that will create others of the same mould. Let us explain by example:

```
H: CONSTANT CREATE ,-T DOES> @ ;
```

The H: tells the Host that this definition is for it and not the Target. All the words between **H:** and **DOES>** will be compiled into the Host as a Host definition. All the words after **DOES>** will be Target words defined in the Target. So what are we doing?

[guru warning]
CONSTANT is the new name that we are adding to our compiler. **CREATE** will create a header in the Host when **CONSTANT** is run at Target compile time. The word **,-T** will save any value on the Host stack into the next available space in the Target dictionary when **CONSTANT** is executed. **DOES>** will terminate the Host definition of **CONSTANT** and enter the compile state for the Target. **DOES>**starts by compiling into the Target a **CALL DODOES**. The code following **DOES>** will then be compiled into the Target dictionary. The address where this compilation takes place is patched into the definition of **CONSTANT** in the Host.

So when we invoke **CONSTANT** in our program, the address of the code compiled by **DOES>** will be compiled into the Target, followed by the value on the Host stack. So when the resulting constant is run in the Target it runs **CALL DODOES**. This leaves the address of the value that was on the Host stack at compile time on the Target stack. The code compiled after **DOES>** is then run. This results in the fetching of the compiled value from the code space onto the Target stack.

The word **CONSTANT** will now create as many more words as we wish, all with the same resultant run-time action, but each with it's own result. Those who know of the Object Oriented Programming style, OOPs, may recognise this construct For more insight into this you may wish to refer to the Appendix.

Note: The example shown here for a constant is not that used by TLM. It is only a simple demonstration of the **CREATE...DOES>** structure.
[end of guru warning]

Demo Application

Loading the Demo

Now we have communication with the TLM we may see what can be achieved. Try the following:

```
 1 2 + . (cr) Error: + is undefined
```

As this is the start we do not have many Forth words loaded yet. To help with interactive development any word in the Library may be loaded from the keyboard with **REQUIRED**, e.g.

```
REQUIRED + (cr) ok

1 2 + . (cr) 3 ok
```

Which shows that the TLM can add 1 and 2 to get 3, providing that the necessary resources have been supplied to it.

All seems to be working so let us jump up to the real thing and try a multi-tasked program with timer driven interrupts.

```
Type: FLOAD LDEMO (cr)

Unresolved references:

*** No words Unresolved ***

Statistics:

Last Host Address: $4AE54

Last Target Code Address: $4B2

Last Variable Address: $2E3

Last EEPROM Address: $0 ok
```

What happened? First we input **FLOAD LDEMO**. **FLOAD** is the Forth word for File Load. It takes the next space-delimited word in the input string as a file name and tries to interpret the contents. It has the same effect as you typing the file contents at the keyboard. These are ASCII text files. You will notice that no file extension was used, **FLOAD** assumes .FTH if no extension is given. So if you use .FTH for all your files then you may refer to them by name only.

The file LDEMO.FTH asks for other files to be loaded to make up the demo program. As each one loads it's path and name appear at the top of the screen.

Running the Demo

Now you have loaded the demo, let's run it. Type:

```
GO (cr)
TTY (cr)
ABCDEFGHIJ 123456789 ABCDEFGHIJ
ABCDEFGHIJ 123456789 ABCDEFGHIJ
ABCDEFGHIJ 123456789 ABCDEFGHIJ
ABCDEFGHIJ 123456789 ABCDEFGHIJ
…
Done
```

The screen should clear and the following appear forever. The result is not too impressive but what is happening is quite interesting. TTY is just a simple terminal emulator. It accepts serial input and puts it on the screen. There are three tasks, described later, that result in these characters appearing.

The first is called **LETTERS**, sends the string '**ABCDEFGHIJ**' to a 'pipe' (See Pipes). The second **NUMBERS**, sends the converted number string '**123456789**' CRLF to the same pipe. The third **UART**, takes one character at a time out of the pipe and sends it back to the Host. The rate at which the characters come back is controlled by a Timer interrupt running every 111mSecs.

So you see the Target is quite busy. The tasks **LETTERS** and **NUMBERS** are endless, they just run and run. **UART** only runs once and then **STOP**s. The Timer interrupt **AWAKEN**s **UART** and it runs once again. So the characters occur at a 111mSec rate.

The funny thing is why do the letters and numbers, that are being put in the same pipe, not get jumbled up and why two sets of letters and only one of numbers with the CRLF after letters and not numbers?

The answers to these and other questions come later. (see Semaphores)

Viewing the Demo

To stop the demo press the ESC key on the PC. Type:

```
?REM (cr)
<- Mega103 Stack Empty ok
```

you should see something like this. We now are back to before running the demo.

We have loaded several new words, the only one you have used of so far is **GO**. IRTC can show you them all in two ways.

The first is to use the ForthEd editor in AIDE. Type:

```
ED DEMO.CTL (cr)
```

The editor screen should appear with the contents of the file DEMO.CTL displayed. This is the so called control file for the demo program.

The DEMO.CTL file does not give us much information about the program, only the files that go to make it up. So open another file to brose that one.

Inside the AVR TLM-Forth

Forth has been used much more widely that many people think. The reason that it is not recognised is that the applications are embedded into products. This package, it is hoped, will continue this tradition.

Inside Forth

A detailed description of the TLM Forth is given here to assist those wishing to use code definitions in particular. The so called Forth inner interpreter, **NEXT**, and the associated call, **NEST**, and return, **UNNEST**, definitions make up the Forth machine.

NEXT

This is the heart of a Forth machine. In a subroutine threaded Forth **NEXT** is a **CALL** instruction. The return address is placed on the return stack by the processor, the address called is called the Code Field Address, CFA.

NEST

Every high level Forth word is a subroutine. It is called and the CFA points to the code. The IRTC compiler keeps track of the Forth words and compiles **CALL** or **RCALL** instructions to call each word used within a definition.

Each Forth word starts with a **:**

EXIT

Every **NEST** must **EXIT**, as every Jump to Subroutine must Return from Subroutine. The value on top of the System Stack is popped by a **RET** instruction that is compiled by the **;** at the end of each Forth word definition.

Data Stack

Forth is a stack based language. The Return Stack used for calls and returns has been described above. The Data that is to be manipulated is placed on the Data Stack. The processor's YREG is used to control this stack. The TLM does however, have an unusual feature. The top Data Stack item is always in the XREG. The Data Stack controlled by YREG

containing any items below this. In a machine code definition the X and Y registers must be saved if they are required by the definition, and restored if the definition itself does not affect the Data Stack. This is most easily done by pushing out onto the Return Stack at the beginning of the definition. For use by the tasks, a USER location, SP0, is reserved for saving the Return Stack pointer, the Data Stack pointer is saved on the Return Stack.

Code and Variable Space

In an AVR embedded system the Code space is in Flash memory. The Variables must therefore have a RAM area for their contents. These two spaces are controlled by different words.

The Code space is set by **ORG**. This is similar to the construct in assemblers.

```
Use: $0060 ORG
```

The address $0060 is set in AVR xxxx.INI. It is the beginning of the Target development code space.

The Variable space is set by VORG.

```
Use: $0060 VORG
```

Tests on the validity of the memory spaces are done at compile time. The following VALUEs may be altered to reflect your memory by editing AVR xxxx.INI:

ROM-START set to $0000

ROM-END set to $FFFF

RAM-START set to $0060

RAM-END set to $03FF

8 Meta Compiling and the TLM

Meta Compiling

Meta Compiling, as its name implies, is the creation of something from itself. IRTC generates a Forth for the Remote Target from the host Forth running on the PC. Meta Compiling is often also called Cross Compiling, as the host and target CPUs are different.

When in the Host mode we Meta compile and create a Target image inside the PC. This image cannot be executed here as the code is for the AVR processor. However, the resultant code may be transferred, Downloaded, to the Target or to an programmer. In both cases the result is code generation that may be run in the Target.

When in the Remote mode we compile and create a Target image in the Host and in the Target. This is slower, as we have two things to do and one is via a serial link. But the result is instant code that may be executed.

Which approach is used depends on the position you are in with your application. To begin with you would experiment in the Remote mode until a section of code was written and debugged. Later you may compile and download the known code and then continue in the Remote mode to debug more.

What happens when we Meta Compile is quite simple, deceptively so. IRTC parses the ASCII text, from a file or the keyboard, and interprets the result. That is every space delimited word is searched for in the vocabularies shown in the right corner of the screen. If found the word is executed in the Host-PC. What happens as a result depends on the mode, Host or Remote, and on a variable **`STATE-T`**.

If we are in Host and **`STATE-T`** is false the word, if found, will execute in the Host-PC leaving its result on the Host stack if necessary. Any Target words will not be found in this state so will result in the message:

```
Error: word is undefined
```

If we are in Remote and **`STATE-T`** is false, the word will execute in the Host-PC and, if a Target definition, its Code Field Address (CFA), will be sent to the Remote Target for execution there also. Any stack result will be left on the Remote Target stack.

If we are in Remote and **`STATE-T`** is true, the word executes in the Host-PC, and if a Target definition, its CFA is sent to the ISP to be placed in the next Target dictionary location. This is the Remote compile mode, set by encountering a **`:`** or **`]`**.

As can be seen the Host executes everything it finds in the Target vocabulary. If **`STATE-T`** is false the Remote Target executes the resulting CFA, if not it compiles it for execution later when the new definition is run.

Any definition found by IRTC that is not Target related just executes in the Host-PC and leaves its result there.

During Target **`CODE`** definitions all the Assembler words are executed. The object code generated is placed in the Target image in the PC and sent to the ISP for storage in the Remote Target, if in the Remote mode. The execution of these words by the Host causing this result

Unresolved References

The statistics from our DEMO load displayed a line for Unresolved References when we ran **`.STATISTICS`**. None were shown for our demo as all the references, words, could be found

when compiling. If this is not the case, say because a word was incorrectly spelled, the word name would appear under the Unresolved Reference heading. IRTC differs from many Forth implementations in that forward references are allowed and are automatically resolved. This applies to Forth kernel words as these are contained in the Library. Any forward reference made in your application code will be resolved when the definition is compiled later from the source. In IRTC it is therefore not usually necessary to explicitly define forward references, although you may with FORWARD:. This is true for your Target application code only. Library extensions do require special conditions for forward referencing and will cause compile exceptions if they occur.

Remote Target - TLM

The Target Link Monitor is a small, 712 byte, program that allows the communication with the Host PC and implements multitasking. This small program is however self contained. It requires no further code to allow full modification and inspection of all the AVRs resources.

This is all carried out by the word **SERVER** which is defined as follows:

```
: SERVER (S - ) BEGIN ADDR@ EXECUTE $A5 CHAR-OUT AGAIN ;
```

We must thank the University of Rostock (Germany) for this deceptively simple idea. The words **ADDR@** waits for the Host to send two bytes of data, LSB first, to the Target stack. This 16 bit value must be a valid CFA, Code Field Address, of an existing Target definition. The word **EXECUTE** performs a IJMP to this address. The **$A5 CHAR-OUT** is used as an acknowledgement for the communications.

The basic TLM words are:

```
ADDR@ C@ C! EXECUTE PAUSE @ ! SERVER SP0 STOP WORD>
```

With only these it is possible for **SERVER** to interrogate the AVR internals.

For example, if we wish to look at the current value of the SREG register, we need to perform a **C@** on file address $3F and send the result to the Host for display. This is what happens:

```
  I/O SREG C@ H. <cr>
```

Host sends CFA of **ADDR@, EXECUTE** runs it.

Host sends $3F (SREG adds) which **ADDR@** puts on Target stack.

Host sends CFA of **C@**, **EXECUTE** runs it and **C@** fetches the value.

Host sends CFA of **WORD>, EXECUTE** runs it and sends result to Host.

Host displays the 8 bit value in Hex.

SP@ puts the value of the stack pointer, DP ,on the stack so using **WORD>** and **C@** the Host may display the current Target stack contents. When in the Remote mode, **.S** does this function.

With **C!** the Host may modify AVR resources. This allows you to write Host programs that require no further Target code but that will exercise hardware external to the AVR. This is very useful in development and production, to do initial testing

9 Library

To make best use of the code space of the AVR we do not compile a Forth kernel with all the standard Forth words, and then compile our application. Instead we use a Library. All the Forth definitions are compiled into the **LIBRARY** vocabulary and executed by the compiler as they are found in the input stream. The definitions in the Library are special but the words **L:** and **L;** allow you to create your own functions to extend these definitions.

Library words are required to perform two differing functions depending on their use. If a word is encountered when compiling a : definition in the Target, it is made into what is called a Forward reference, if it has not been previously compiled. When the compiler reaches the end of the current definition it tests the **TARGET** vocabulary for any such references. These are then looked up in the **LIBRARY** vocabulary to see if they exist. If so the Library definition is executed. This will generate the Target code necessary and resolve the references. If, however, the Library definition contains other Library words their execution must create a forward reference, for now, that will be resolved later. This entire process continues until no more unresolved references exist in the **TARGET** vocabulary that have equivalents in the Library . As Library words may contain other Library words it is necessary that no forward references exist in the Library. The compiler will give an exception to any word it finds that has not been previously defined.

You may add to the Library both assembler and Forth definitions. These are defined as follows:

High Level:

```
L: <name>
  word1 word2 word3
  EXIT
L;
```

Code:

```
L: <name>
  M[ ADIW TOS , # 2
  INC TOSL
  PUSHT
  RET ]M
L;
```

In the High Level definition, word1,2 and 3 must exist in the Library prior to compiling this new definition. M[and]M must start and end any code fragments.

Note: High Level code must end with the **EXIT** and Code Routines end with **RET** or a **JMP** to code ending in **RET**.

Also: If you wish to use a Forth word from the Nucleus like, **BRANCH**, **?BRANCH**, **PAUSE**, **STOP**, **STATUS**, **BASE**, **PTR** or **PAD**, these must be preceded by **[TARGET]** in the Library definition. This is because they do not have a Library version. Also any fixed value must be followed by **LITERAL** or **DLITERAL** to give the correct compilation.

Number Conversion

The AVR Library contains the fundamentals of number conversion. That is, ASCII string to a number on the stack and a number on the stack to an ASCII string.

The strings are held in an area called **PAD**. This traditionally has been 64 bytes beyond the current end of the dictionary. In embedded systems this is not practical for two reasons:

- Dictionary is in memory when running.
- Not re-entrant for tasks.

To overcome these **PAD** has been defined as a **USER** variable. It is 11 bytes above it's nearest partner **PTR**. This is to allow for the number to ASCII string to be converted down from **PAD**, the largest double number is 2,147,483,648 or 10 characters plus a sign makes 11. The string for conversion to a number is traditionally from PAD up, so enough space above is also required.

The definitions in the Library are only the conversions. **NUMBER** takes the address of an ASCII string and returns a double number and a true flag if sucessfull. **(D.)** takes a double number and returns the address and length of the converted string. Both require the **USER** variable **BASE** to be loaded with the radix required for the conversion. These are then used by application dependant code to input and output the strings.

These conversions are for double numbers, you may use them as such, dropping the top item for 16 bit, or modify then for different applications.

10 Multi-tasking

Multi-tasking is a function that until recently was not so generally known. The Forth community have been using it for many years, indeed Charles Moore's first Forth's used it. Once used you will wonder how you did without it. The version in the TLM, there are many, is the simplest and called a Round Robin tasker. Basically each task is a separate program that is often self contained. The tasks run until they reach a point where they have to wait for something, at this point a task **PAUSE**s and the next runs. If the task does not have a natural wait then a **PAUSE** must be added at some convenient point, or the task **STOP**ped. These tasks are said to be co-operative, that is they must relinquish control of the processor to allow others to have a go. Each task has it's own stack space and an area called **USER**. Here resides the links to the tasks memory plus variables that may be used only by that task. One example of these is **BASE** used by the numbers, described elsewhere. A task is given a name and the sizes of the Data stack, Return stack and User area these are then allocated to it by **TASK:.** The stacks work down from their tops and the user area goes up, as shown below.:

USER **Area**

The **USER** area starts with the address of the next task area to link to and is called **STATUS**. The other bytes defined by IRTC are show below:

Name	Location	Function
STATUS	UP+0	The link to other tasks and the **REST**ing, **AWAKEN** flag.
DP0	UP+2	The address of the bottom of the Data stack.
HANDLER	UP+4	The exception return address for **CATCH** and **THROW**.
VFLAG	UP+6	The **TO** variable flag.
BASE	UP+8	The radix for number conversion.
PTR	UP+10	The pointer for number to ASCII conversion.
PAD	UP+24	General area used by number conversion.
	UP+34	Next available USER address.

Running Tasks

The Task areas allocated by **TASK:** which is preceded by three values to set the task data, return and user stack areas. Then we must chaine the tasks together at runtime. To do this we **LINK** the tasks.

```
task-name LINK
```

The **LINK** process adds a task to the chain but sets the task as **REST**ing. To make a task run the application we wish, the **USER** area and the task's stacks must be set to run whatever word we select. The task must be **ACTIVATE**d.

```
task-name ACTIVATE <word>
```

This must be inside a **:** definition. **<word>** must be an endless loop or terminate with **STOP**.

Semaphores

To keep tasks synchronised we use semaphores. These are global variables that are loaded with a unique value relating to the task that wishes to take control. This value is the Task's **USER** area address. A semaphore is initialised to zero. At this point any task may take control. When a task wishes a resource it **ACQUIRE**s the semaphore. From now on no other task may acquire this resource as the semaphore is non zero. To allow others access the task it must **RELINQUISH** the resource and reset the semaphore. This mechanism provides for simple inter-task communication, provided the tasks do not hog any given resource. Tasks must co-operate.

If you refer to GLOBAL.FTH from our earlier demo, you will see **(STREAM)** is used as a semaphore for the 'pipe' **<STREAM>.** Each task acquires the pipe in turn to place it's ASCII string. As the pipe is shorter than the strings the semaphore is necessary to stop the strings getting intertwined. Only when **(STREAM)** is zero may a task **ACQUIRE** the pipe. From then on only that task may use the resource.

11 Pipes

An extension to **TO** variables has been added to the TLM. These are byte pipes, or byte oriented First In First Out, FIFO, structures. These are particularly useful when using tasks, refer to the demo. These pipes may be up to 256 bytes long and are used as TO variables. Values sent **TO** the pipe are placed in the next available byte. If the pipe is full the task is **PAUSE**d until the pipe is free. If a task tries to take a byte from an empty pipe, the task is **PAUSE**d until there is a byte to take.

If a pipe only links two tasks there is no need for a semaphore. Only if more that one task takes or puts data into a pipe is the semaphore required.

To define a pipe use:

```
10 BYTE-PIPE <name> <cr>
```

Before a pipe may be used it must be initialised. The size of the pipe and it's address in the variable space are saved when it is defined. To initialise it the address of the definition is required. A **'**, or **[']** if in a colon definition, will provide the address for **INIT-PIPE**.

```
' <name> INIT-PIPE <cr>
```

or within a colon definition

```
['] <name> INIT-PIPE
```

12 Host Programs

It is possible to automate the interactive command line testing. The TLM is a slave to IRTC and as such we may send a sequence of operations to it provided each will execute before a communication timeout occurs.

If we wish to test the Target hardware before embarking on the software we may set port bits and read port bits from Host programs e.g.

```
ONLY META DEFINITIONS ALSO FORTH ALSO

H: TEST (S -- )
  $01 I/O DDRA [TARGET] C!
  BEGIN
    10 MS  1 I/O PORTA [TARGET] C!
    10 MS  $FE I/O PORTA [TARGET] C!
  KEY? UNTIL
  KEY DROP
;
```

IN-META

This routine will toggle PORTA bit 0 approximately every 10 milliseconds until a key is pressed on the Host PC. **`[TARGET]`** is required to compile the Host address of C! as this is a Target definition.

The **TEST** word above only exists in the Host. It uses the code in the TLM to operate but does not add any extra. It is possible to incorporate Target definitions with the Host ones and then refer to them from within the Host words.

The Host program should be compiled in and run from the Remote mode after communication with the ISP is established.

13 Exceptions CATCH and THROW

Like many high level languages Forth does not have an explicit GOTO. This means that it is often necessary to create proprietary solutions to errors that occur in running embedded code. To help with this the CATCH and THROW words have been added to the AVR Forth to assist with error conditions.

CATCH
The **CATCH** word takes a CFA from the data stack and executes it after first placing its position and the current stack depths onto the return stack. The word **HANDLER** records the position of this stack frame for later use by **THROW** and its previous value is also saved to allow nested **CATCH** and **THROW** structures.

```
' <word> CATCH ERROR !
```

CATCH allows a value to be returned to show the error condition. If no errors are encountered <word> finishes normally and returns to after **CATCH**.

THROW
When an error is encountered in the word run by **CATCH** a non-zero value is placed on the data stack and **THROW** executed. **THROW** takes the value in **HANDLER** and restores the stacks to where they were when **CATCH** was executed except for the addition of the non-zero value given to **THROW**.

```
?DUP IF THROW THEN
```

HANDLER is a **USER** variable in multi-tasking systems and so the error recovery may be used within each task.

14 Turnkey Operation

At some point in your application development it is necessary to run the code directly from Reset. There are two ways this may be done;

`Reset Vector` - Compile a JMP to your application at address $0000.

`EEPROM Vector` - Use **`RUN`** <name> to set the EEPROM start vector and run the code.

The first is the way that a final system should be compiled. The second is useful in testing as the vector may be reset to $FFFF with;

```
END-RUN
```

after the operation has been tested but without having to **`REPROGRAM`**. Also **`RUN <name>`** may be used to test an individual section of your code. The word used by **`RUN`** would be compiled to set any variables, ports, etc. and would then execute your code. RUN places the address of this test routine into $01,$02 of the EEPROM, tri-states the SCK, MOSI and MISO pins from the programmed and Resets the Target CPU. When reset the TLM initialises the Target and runs the test routine from the vector placed in the EEPROM. The **`RUN`** command invites you to press the ESC key to exit. This resets the EEPROM vector and does a **`?REM`** to get back to interactive development. Any other key will exit without resetting, which will leave the code in a turnkey mode. Any **`?REM`** or reset will run the test code until the EEPROM vector is reset.

`IS-APPLICATION` performs the same function as **`RUN`** but also creates a JMP to the application code address at code address $0000.

```
' <application> IS-APPLICATION
```

This should be placed at the end of your code to create a turnkey system.

NOTE: It is necessary for the application code start to set up the **`MAIN-TASK`** and Forth registers before running the high level code. See **`MAIN`** in TLM.

NOTE: It is possibe to get into a lock-out situation if your test or application code uses the development UART port, as with the muli-tasking demo. Here the ISP will not enter the program mode because there are characters being sent back by the running application. To fix this use the following;

```
Select the HOST mode, Ctrl+Shift+H.

Type DESELECT


Type TTY
```

any characters in the buffer will be displayed and the ENTER key should produce a ?. Exit **`TTY`** with the Esc key.

```
Type ASCII e PCH CR? SELECT
```

This will erase the Target device and you may recover by REPROGRAMing the TLM only.

15 AVR Assembler

Introduction

The AVR assembler in the IRTC will allow you to do most that a conventional assembler will, and something's that it may not. It is a single pass interpreted assembler with conditional statements, such as

```
IF … THEN
IF...ELSE...THEN
BEGIN...UNTIL
BEGIN...WHILE...REPEAT
BEGIN...AGAIN
```

The syntax of the assembler is not that of Forth, Reverse Polish or Postfix notation but standard Prefix. This means that the source looks like standard assembly but with space delimiters. e.g.

`LDI R26,0x01` becomes `LDI TOSL , $01`

As with all assemblers this checks the validity of the addressing mode for the instruction, and the size of the data. This is accomplished by flags that are set to indicate the various addressing modes.

Note: The only forward referencing available in the assembler is with the structures above and the skip instructions of the AVR

Conditional Flags

The conditionals require a flag test, **CS**, **0=**, **0<**, **0>**, **>=**, **HS**, **TS**, **OV** or **IE** before their use. Each may be followed by a **NOT** to produce the inverse of the test. e.g.

```
0= IF       \ branch if non zero.
0= NOT IF   \ branch if zero.
```

Register Use

The AVR register use is described in the TLM-Forth section. Here is a résumé of that usage:

YReg - Data Stack Pointer
XReg - Top of the Data Stack
SP - Return Stack Pointer
R22,23 - UP, Task Area
R16-21 - N, ScratchFile Use

Macros

Because of the way Forth and the Assembler work it is possible to define Macros for use in the assembly process. If we make a Host definition which contains assembler words, when interpreted at compile time, the definition will execute these words. This will place the assembler code directly into the Target. An example of this technique is:

```
XASM DEFINITIONS

MACRO: PUSHT
  ST Y+ , TOSL
  ST Y+ , TOSH
MACRO;
```

```
IN-META
```

Using this technique it is possible to construct repetitive code for insertion in your machine code programs

Note: You may also produce the same result with an **H:** definition and **M[..]M** e.g.

```
H: PUSHT
  M[ ST Y+ , TOSL ST Y+ , TOSH ]M
;
```

PUSH and POP

Four other macros are pre-compiled to assist with stack control.

PUSHT places the TOS contents onto the data stack with the following code;

```
  ST -Y , TOSL
  ST -Y , TOSH
```

POPT gets the data stack into TOS whith the following;

```
  LD TOSH , Y+
  LD TOSL , Y+ ;
```

PUSHS places the SEC contents onto the data stack with the following code;

```
  ST -Y , SECL
  ST -Y , SECH
```

POPS gets the data stack into TOS whith the following;

```
  LD SECH , Y+
  LD SECL , Y+
```

If you wish to manipulate the stack you should do so with these macros. If you write in-line code and use these macros the optimiser will try to elliminate unnecessary stack use.

Addressing modes

The AVR has many and varied addressing modes. By no means all are used in the TLM, but many are. To understand their use it is strongly recommended that you read the ATMEL data book for a full explanation. Here we will consider the addressing mnemonic used in the IRTC assembler.

Register Syntax

The same syntax that appears in the ATMEL data book has been used for register identification. A upper case 'R' for a register, 0-31, and a upper case 'P' for a I/O register, 0-63. Register pairs are indicated by 'RR' and only apply to R24, 26, 28 and 30. For convenience and improved readability the registers, register pairs and I/O registers are pre-defined, R4, X, Z+, -Y, Z+#, BIT0 and SREG etc. These may all be constructed but with Forth space delimited format e.g.

```
  4 R 26 RR 0 BIT 63 P
```

The pre-defined versions are much more readable. Also all the I/O registers and the flags within them are predefined by name e.g.

```
  SIB UCR , RXEN     \ Will enable the UART reciever.
```

Note: When using the I/O Registers in a high level definition they must be preceeded by I/O. e.g.

```
$51 I/O PORTB C!
```

Note: When using the Register flags in a high level definition they must be preceeded by **FLAG** to generate a mask e.g.

```
FLAG CS00 I/O TCCR0 ON
```

To differentiate between destination and source operands a space delimited comma is used e.g.

```
ADIW TOS , 2
```

BEWARE: of the use of a Forth **,** while in the assembler as very odd results will occur.

Immediate

The operand value used by the instruction is the value supplied by the operand field itself. The hashmark (#), optional, is used to distinguish data from an absolute address and may only be in the source field.

Examples:

```
LDI R26 , # 04        \ Adds 4 to the value originally
                      \ contained in register R26.

SBIW TOS , # 3        \ Subtracts 3 from the top stack item.
```

Register Direct

In this mode a register is be addressed by using its absolute address in the register file.

Example:

```
ROL R4                \ Rotates the contents of
                      \ register number 4 left one bit.
```

Data Indirect

In this mode the address of the data does not appear in the instruction, but is located in a register pair, X, Y or Z.

Example:

```
LD TOSL , X
CLR TOSH
```

This is equivalent to a **C@** in Forth, X being the top stack item holds the address of the data byte.

Data Indirect with Displacement

The indirect working register acts as a base or starting value to which is added an immediate offset, 0-63, to point to the data. The offset value is the immediate value given in the instruction while the index value is given by the contents of a register pair.

Example:

```
LDD R10 , Y+# 5       \ If Y contains 55 then the
                      \ contents of 60 (55+5) will
                      \ be loaded into register 10.
```

Data Indirect Post-Increment

Here the destination or source addresses are given by the contents of a register pair, X, Y or Z, which are then post-incremented.

Example:

```
LD TOSH , Y+
LD TOSL , Y+
```

In IRTC Y contains the Data Stack pointer, DP. The above example is equivalent to a DROP.

Data Indirect Pre-Decrement

This is similar to Data Indirect with Post Increment, except that the indirect register pair is decremented BEFORE the data is accessed.

Example:

```
LD R7 , -Z
```

Direct Bit

In the Direct Bit addressing mode, any bit in any register or I/O can be addressed and potentially modified.

Examples:

```
SBR R7 , BIT3         \ sets bit 3 in working register 7.
CBI PORTC , BIT4      \ clears bit4 in I/O portc.
```

Data Direct

This mode addresses the specific location within the data memory directly. It only needs the absolute address value.

Example:

```
STS COUNTER , R9      \ location COUNTER is loaded with
                      \ the contents of the register 9.

LDS R9 , COUNTER      \ R9 is loaded with the contents
                      \ of the data memory location COUNTER.
```

Code Memory Indirect

This mode uses the Z register pair to hold the absolute code memory address.

Example:

```
LPM                   \ The code memory addressed by Z
                      \ is loaded into R0 only.
```

Note: The address is the byte address, the LSB specifies the high or low byte.

Direct Program

The program execution continues at the address contained in the instruction.

Example:

```
 JMP ' PAUSE
```

Jumps directly to PAUSE. The ' is used in Forth to find the CFA of the following word. If the word was created by a LABEL the ' is not required.

```
  CALL ' STOP         \ calls the subroutine STOP.
```

Indirect Program Addressing

The program execution is continued at the address contained in Z.

Examples:

```
IJMP                \ Jumps to the address in Z.
ICALL               \ Calls the address in Z.
```

Relative Program Addressing

The offset, -2048 to 2047, is held in the instruction causing the execution to continue from the Program Counter plus the signed offset.

Example:

```
RJMP ' TEST         \ Jumps to TEST relative
RCALL ' TEST        \ Calls TEST relative
```

New Conditional Opcodes

The AVR instruction set has four skip instructions for bit testing in the register and I/O space. New opcodes have been created to make use of these to reduce code and increase speed in conditional assembly. The new opcodes are;

IF-BRS - Skips one instruction if bit in register set

IF-BRC - Skips one instruction if bit in register clear

IF-BIS - Skips one instruction if bit in I/O set

IF-BIC - Skips one instruction if bit in I/O clear

SBAK - Skips back one instruction

The IF- instructions are combined with **ELSE** and **THEN** to create conditional assembly. They may also be used instead of **WHILE** in a **BEGIN**...**WHILE**...**REPEAT** structure. The **SBAK** may be used to create a fast loop waiting for a bit to set or clear. See **(OUT)** in the AVRPROG3 file.

Code Definitions

To define a Forth code word the assembler must be invoked to allow the instruction words to be found. Three words will do this; **CODE**, **LABEL** and **INTERRUPT**.

CODE creates a Header for the word following and causes the assembled machine code to be run when the word is executed. This is just like a Forth **:** definition only in code.

LABEL creates a Header for the following word, that only leaves the address of the assembled machine code on the Host stack when the word is executed. This is often used as a reference for jumps or branches within a **CODE** definition.

All assembler definitions must end with **END-CODE** or **C**;. These words terminate the assembler and test the Host stack for irregularities. **CODE** definitions will usually end with **RET**.

In-Line Code

As AVR Forth is subroutine threaded and produces machine code, it is possible to put code fragments into a high level definition. The code is encompassed with C[and]C, for example;

```
: TEST C[ ADIW TOS , 4 ]C ;
```

This adds 4 to the item on top of the stack.

CODE Stubs

If you are only using assembler code you may still use TLM to test your routines.

TLM expects the CFA it receives to be that of a subroutine. If you have used **CODE** definitions for your routines these may be executed by name from the command line. If they are defined by **LABEL**s then a stub is required e.g.

```
CODE TST JMP ' <name> C;
```

This assumes **<name>** is a routine ending in a return opcode and that it does not corrupt the TLM file space or stack.

To test parts of your code it may be necessary to insert return opcodes into the code at strategic points and then create stubs to test that section.

Interrupts

Interrupts in the AVR are done via vectors in the first 48 bytes of the program memory space. To enable these vectors to be changed during development the TLM has 96 locations programmed to jump indirectly to locations at the start of the development code RAM. These RAM locations may then be programmed with an absolute jump address to the interrupt routine. This adds about 3.5uSecs to the interrupt latency at 6MHz. The Forth word **VECTOR!** takes a vector number, 1-24, and the interrupt address to automatically create the jump in the RAM location.

To return from a RAM vectored interrupt you should use the **VEC-RETURN** function that pops the N, ZL, ZH, and SREG that were saved as part of the (VECTOR) execution.

Note: This is only possible in the larger AVR devices with **JMP** instructions, that allocate 4 bytes for the interrupt vectors from $0000. In the smaller devices it is necessary to change the interrupt vector and **REPROGRAM**.

The word **INTERRUPT** may be used to define an interrupt routine instead of **CODE** or **LABEL**. The associated interrupt vector address must preceed INTERRUPT and it will create a **JMP** or **RJMP** to the routine in the code vectors. This will only take effect when the whole image is **REPROGRAM**ed. The function **VEC-SAVE** may be used to push N, ZL, ZH, and SREG registers so that the **VEC-RETURN** may pop correctly.

```
  TIM0-OVF INTERRUPT <name>
    VEC-SAVE
    ... interrupt code …
    JMP VEC-RETURN
  C;
```

Or

```
  END-INTERRUPT
```

During development the above interrupt would be coded as;

```
CODE <name>
\ TIM0-OVF INTERRUPT <name>
```

```
\ VEC-SAVE
  ... interrupt code ...
  JMP VEC-RETURN
C;                          \ or END-INTERRUPT

: SET-TIM0-OVF-INT
  [ XASM 17 ' <name> ] LITERAL LITERAL
  VECTOR!
;
```

The **SET-TIM0-OVF-INT** is run during the initialisation to set the RAM vector. The interrupt may then be tested and when operational the **CODE <name>** replaced by the **TIM0-OVF INTERRUPT <name>** and **VEC-SAVE** . If your interrupt code may be coded with only the use of the N, ZL, ZH, and SREG registers you do not need to save any other resourses.

Dumps

To assist with code definitions in particular, memory dump utilities **DUMP**, **RDUMP**, **IDUMP**, **FDUMP** and **EEDUMP** are available. **DUMP**, **FDUMP** and **EEDUMP** require a start address and the number of bytes to dump.

DUMP is for the code memory e.g.

```
  $0000 50 DUMP
```

Register File Dump

The register file space may also be dumped by using **REG-DUMP**.

This shows the contents of all the file registers from R0 to R31.

I/O File Dump

The I/O file space may also be dumped by using **IO-DUMP**.

This shows the contents of all the I/O registers from 0 to 63.

Data Memory Dump

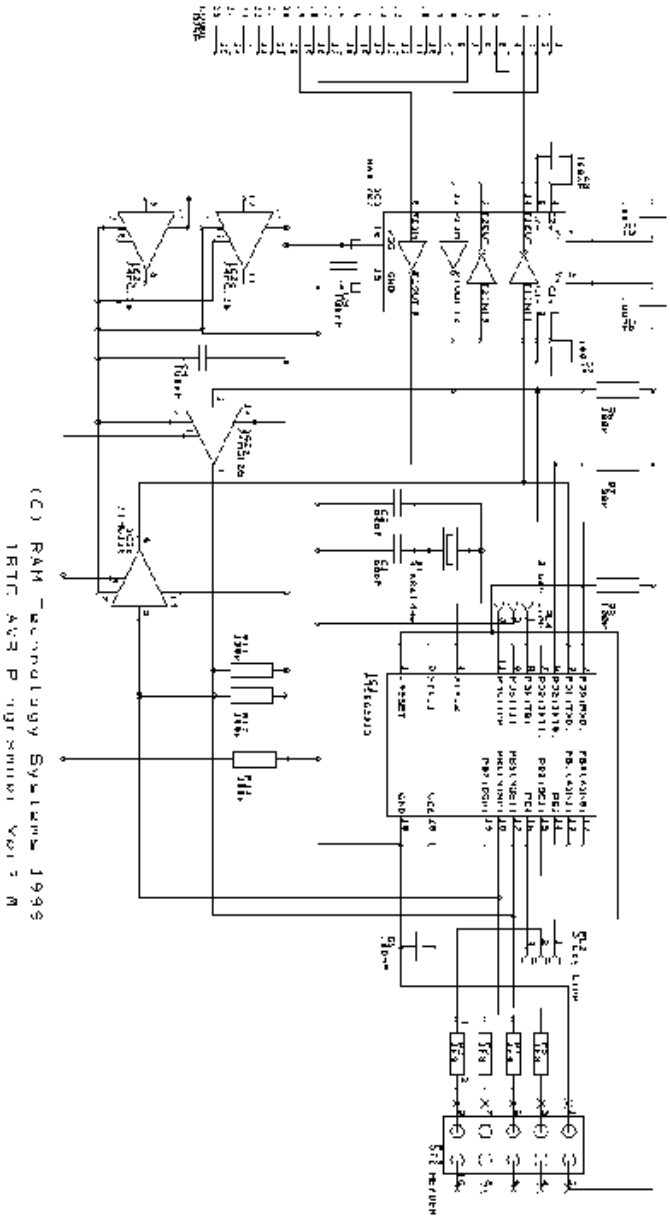
The Data memory may be dumped by **FDUMP**. As the registers and I/O are also in the Data Memory you may use this to do part of the dumps shown by **REG-DUMP** and **IO-DUMP**.

E2PROM Memory Dump

The data in the E2PROM may be dumped using **EEDUMP**.

16 Interactive Serial Programmer

Circuit



The IRTC ISP is similar to the Atmel serial programmer explained in their application note. However, a tri-state buffer has been added to the IRTC ISP to enable the programmer to be used for both programming and development.

The control codes are the same as in the Atmel application note with the exception of Z. This leaves the ISP in the program mode but tri-states the SCK, MOSI and MISO lines and sets RESET high to enable the target CPU to run stand-alone.

The DTR line on the serial port controls the ISP operation, switching between program and development mode. In the program mode the 90S2313 on the programmer accepts ASCII commands to control the program functions. These functions are the same as the Atmel ISP. In the development mode the 2313 is switched off and the buffers enabled allowing the Rx and Tx signals to pass through to the MOSI and MISO lines. The TLM receives the serial data and enables development.

Header Signal Connector 5+5 way

The ISP is powered from the target system and requires the following connections to the target processor;

Pin1 - Black - GND 0 Volts
Pin2 - Red - VCC +5 Volts
Pin3 - Blue - MOSI Master Out Slave In
Pin5 - Green - MISO Master In Slave Out
Pin7 - Black - SCK Serial Clock
Pin9 - White - RESET Target CPU Reset

The ISP must also be connected to the host computers serial port, COM1-4. It may be necessary to use a 25Way to 9Way convertor or cable to enable proper connection.

The Target and the ISP should be powered up together to ensure they reset properly. It is also important that the ISP is connected to an operating serial port with PC running prior to power-up.

Checking the fuses

It is possible to get the Target CPU out of step with the ISP. This may be caused by noise, especially on the SCK line or bad earthing of the PC and power supplies. In some cases the signature bytes may read as $FF. If this occures check the fuses with **?FUSES**, this should return $0F. If not use;

```
$0F FUSE!
```

to correct. **?REM** should now return the proper signature bytes.

Signatures

The AVR chips all have a 3 byte signature. For example the 90S8515 is $01 $93 $1E. These are set in the value SIGNATURE in the .INI file. When you do a ?REM the Target chip signature is checked against the set value. If they do not agree IRTC will not allow remote access. In the case of the Mega103 the bytes should be $01 $97 $1E but in some devices they are $01 $01 $1E. If you have some of these you may correct for this by editing the Mega103.INI file.

Commands

A SET-ADDRESS - Sets the address to be programmed next.

C WRITE-PROGH - Write a code byte high.

c WRITE-PROGL - Write a code byte low.

D WRITE-DATA - Wite a byte to EEPROM at address set.

d READ-DATA - Read a byte from EEPROM.

e CHIP-ERASE - Erase the Code and Lock bits.

F READ-F&L - Read the Fuses or Lock bits.

f WRITE-FUSE - Write the Fuses.

L DEV - Put the programmer in the Development mode, RS232 pass thru.

l WRITE-LOCK - Write the Lock bits.

m WRITE-PAGE - Write a 256 byte page, Mega103/603 only.

P PROG-ENABLE - Enable the program mode, returns $53 if successful

p PROG-TYPE - Returns 'S' to show a serial programmer.

R READ-PROG - Reads the Code memory at the address set.

S IDENTIFIER - Returns 'RAM ISP' to show programmer.

s READ-SIGNATURE - Returns the signature byte at the address specified.

T GET-DEVICE - Sets the device code for AVR to be programmed.

V SOFT-VER# - Returns the software version.

v HARD-VER# - Returns the hardware version.

Z TRI-STATE - Tri-states the programmer pins and sets reset high to run the Target.

: GENERAL - Sends the byte received out of SPI and returns the SPI byte clocked back.

17 Object Save and Load

IRTC, when in the Host mode, will compile an application very quickly. However, when it has been written and debugged, it may be desirable to save the resultant code.

This may be done with **HEX-SAVE**.

The start and end addresses are calculated and given to **HEX-SAVE** with a filename. The file extension .HEX is automatically appended to the filename.

```
Use: $0000 $0FFF HEX-SAVE APPL
```

The code is saved in the INTELLEC format, and may be copied to a programmer.

Saved code may be re-loaded into the Host Target image with **HEX-LOAD**. Again no file extension is required.

A * is shown for every 1k of code loaded, and a checksum of the code is given at the end.

Note: The addresses used by **HEX-LOAD** and **HEX-SAVE** are byte addresses not word. To save your total application use;

```
  0 HERE-T HEX-SAVE APPL
```

Also the word **CODE-FILE** will create a .HEX file called with the code range 0 to **HERE-T**. This may be viewed by the Atmel Studio debugger AvrDebug.exe. The word **SEE** will try to find the following word in the Target and if compiled shows the CFA of the word, creates CODETEST and runs AvrDebug. For this to work it is necessary to copy AvrDebug.exe into the COMPILER directory.

18 External Programmers

There are three methods of transferring code to an EPROM programmer for Production code generation.

From within IRTC two words **BIN>PROG** and **HEX>PROG** allow a binary or an ASCII Hex download via COM1:. Both these require the code start address and end address.

```
COMMS 1 8 NONE 9600 BAUD
$0000 $03FF BIN>PROG
$0000 $03FF HEX>PROG
```

The Binary has no checksum or address, it is just a dump of the code at the addresses from the Target image in the Host.

Similarly the Hex is just a code dump, again with no checksum or address transmitted.

Both give a checksum, on the screen, for the bytes sent for comparison with that generated manually in the programmer.

The last method is to use the INTELLEC file appl.HEX. This contains records with address and checksum. To download this to the programmer MS-DOS COPY may be used. This may be from the command line of DOS or from within IRTC with:

```
SHELL COPY appl.OBJ COM1: /B <cr>
```

The word **SHELL** may be used to send any combination of DOS commands from within IRTC. **SHELL** alone will shell out, returning via **EXIT**.

19 Errors

Address NOT in the ROM

The dictionary pointer DP-T has exceeded the value in ROM-END.

Address NOT in the File

The variable pointer VP-T has exceeded the value in RAM-END.

Address NOT in the E2PROM

The E2PROM pointer has exceeded the value in EEPROM-END.

Already a GOTO!

Compiler was converting a call to a goto and found it was already.

Already Resolved

The word being resolved is already resolved. Maybe a duplication.

Conditionals Wrong

The compiler has detected that the conditionals are not matched. An **IF** may not have a **THEN** or a value may have been left on the Host stack during compilation.

Definition out of range!

This only applies to devices with more than 2K of code space. A jump within an **IF..THEN**, **BEGIN..UNTIL** etc. is more than a relative jump.

Not enough Parameters

The word being executed by the Host did not have enough parameters on the data stack to run.

NOT Erased

The address shown is not $3FFF.

NOT Compiling!

The compiler is interpreting and should be compiling.

NOT an 8 bit number

The value on the Host stack is greater than 255.

NOT in Remote!

IRTC is not in the Remote mode and should be.

NOT Resolved!!

The word is not yet resolved and still has forward references.

is NOT a Library Definition!

The word could not be found in the Library vocabulary.

is NOT yet Defined or is In-Line!

The word does not have a header in the Target vocabulary so it may have been used but only produced in-line code.

Could NOT enter Program Mode!

The ISP could not establish contact with the Target CPU. Reset and try again.

Prog. Error!

The ISP did not respond to the command properly.

.... Unresolved Word(s)

Shows the words that are unresolved.

Error: <word> is undefined

The word could not be found in the current search order

20 Suggested Reading

A range of Forth books is stocked by MicroProcessor Engineering. Among them, these are suitable introductory books.

Starting Forth - Leo Brodie - ISBN 0-13-842922-7

Object-Oriented Forth - Dick Pountain - ISBN 0-12-563570-2

Thinking Forth - Leo Brodie - ISBN 0-13-917568-7

21 Further Information

Forth Interest Group UK http://www.users.zetnet.co.uk/aborigine/forth.htm

Forth Interest Group USA http://www.forth.org/fig.html