Appendix B

FORTH++ and the MACH1

The *MACH1* is an *RTX-2000* board that can plug into an IBM PC or compatible and communicate with the IBM PC via a 16 KBytes of dual port RAM. It has up to 128 KBytes of fast static RAM. The board's layout is very simple due to the use of a configurable gate array to hold all of the bus interface logic.

The FORTH++ development system is a segmented memory model FORTH which runs on the IBM PC and on the Harris *RTX-2000* family of FORTH processors. It is a FORTH-83 system that includes support for argument records, multi-tasking and windows.

This annex describes some features of the FORTH++ system and the multiprocessor programming environment it provides for an IBM PC with one or more *MACH1* boards.

B.1 The *MACH1*

The *MACH1* board, from MicroAMPS Ltd., is fully plug-compatible with an IBM PC expansion slot and is designed to be compatible with existing FORTH development systems. Unlike most other FORTH boards, it also dedicates about 13 square inches of board area (approximately $\frac{1}{3}$ of its full size) to a hardware prototyping area. An uncommitted backplane connector permits use of a DB-25 or similar connector to communicate with any other special equipment.

The Harris *RTX-2001A* is the board's standard microprocessor operating at 8 or 10 MHz and can be combined with 32–128 KBytes of 1- or 0-wait-state SRAM (Static Random-Access Memory). The minimum 8 MHz RTX can deliver bursts of 50 MIPS and sustained operations at 12 MIPS; the faster 10 MHz chip can deliver sustained rates of 15 MIPS.

Existing FORTH cross-compilers using the FORTH-83 and *Poly*FORTH standards are fully compatible with the *MACH1* board. The FORTH++ system was designed to be used with the board and is now distributed with the board as part of a development package (Marriot 1989a).

B.2 The *MACH2*

The *MACH2* board has *two* RTX chips operating in parallel. The board is too large to be mounted inside the IBM PC. The FORTH++ system is used to program this board and is distributed with the boards (Marriot 1989b).

[0]**This is a chapter taken from the Ph.D. thesis "Practical and Theoretical Aspects of Forth Software Development". A full copy of the thesis is available from the British library. Both the this chapter and the thesis are Copyright © Peter Knaggs.**

B.3 FORTH++

The developers of the *MACH1* sent us a prototype board asking us to develop a version of the FATWIN FORTH system to operate on the new board. They had heard of our interest in the RTX-2000 (see Chapter 1) and had previous experience with the FATWIN system, thus the invitation.

We redeveloped the FATWIN system for the RTX environment, in addition to extending the IBM PC based version (Stoddart 1990b; Stoddart 1990c; Stoddart 1990d), which became known as FORTH++.

B.3.1 Memory Organisation

Both FORTH implementations are based on a segmented memory model with native code, names, strings and stacks being held outside of the 64 KBytes of FORTH-83 Standard FORTH memory. In addition, the name and string segments of the RTX FORTH system may optionally be held in the IBM PC's memory so that on a 128 KByte memory *MACH1* board there will be 64 KBytes for RTX code and 64 KBytes of Standard FORTH memory space. The `HERE` (the next free dictionary location) on a fully featured RTX FORTH++ system configured in this way is at `0700` (under 2 KBytes of kernel data space).

This organisation allows large applications to be developed especially since the IBM PC FORTH++ system has a C library interface, graphics and floating point libraries (see Chapter 3).

B.3.2 Multi-Tasking and Windows

Both systems support classical FORTH multi-tasking using a round robin scheduler with non pre-emptive task switching. We call the concurrent objects in our system "actors", but they are functionally similar to *Poly*FORTH's terminal tasks with some additional abilities for passing activation messages.

Due to the RTX using hardware stacks, it can present efficiency problems when switching between tasks. We have found techniques which ameliorate this situation. For example, delayed tasks are handled by the system timer interrupt routine. A system with one executing task and any number of delayed or idle tasks will have virtually no multi-tasking overhead. In addition, interrupt routines can be written in high level FORTH with all the support provided by the argument records mechanism described in the next section. The efficient RTX multi-tasking involves more use of interrupt routines and an absolute avoidance of such indulgences as polling.

The IBM PC version of FORTH++ supports text windows which can be connected to actors. These windows may be opened, overlayed or closed, but in any of these states they can be written to, with a minimum of processor overhead. It is a relatively simple matter to allocate windows to *MACH1* actors. This just needs a new target-to-host mailbox to be defined, an actor on the IBM PC to read mailbox characters and display them in the window and the vectoring of the *MACH1* actors `EMIT` routine to output to the new mailbox.

B.3.3 Argument Records

The distinguishing feature of FORTH++ is its use of a frame stack for local variables. The mechanism is similar to that used by compiled languages such as PASCAL or C. The basic idea is similar to the `LOCALS` wordset currently being proposed for the ANSI-FORTH Standard (ANSI 1991) and is much more efficient and expressive.

As an example of a problem which is slightly awkward to code in classical FORTH, consider the word `SURFA` which calculates the surface area of a cuboid from its length width and height. A definition of `SURFA` using argument records is given in figure B.1. An example test for this definition in given in figure B.2.

In this definition, the immediate word `{` commences the description of an argument list. The immediate defining word `const` is used to create the temporary dictionary entries `length`, `width` and

```
: SURFA ( length width height -- area )
  { const length  const width  const height }
 length width *  length height *  width height *  +  +  2*  ;
```

Figure B.1: A definition of `SURFA` using argument records.

```
10 20 30 SURFA . [Enter] 2200 ok
```

Figure B.2: Testing the `SURFA` definition.

`height`. The argument list is terminated by the immediate word `}`, will compile the code required to move three values from the FORTH stack to the frame stack.

When compilation reaches the end of the definition, the words `length`, `width` and `height` are removed from the dictionary and a run time operator is compiled that will remove the current frame from the frame stack and perform the exit function.

As a second example, consider a word `MAX-OF` that finds the maximum value in a table. Figure B.3 shows a possible definition for `MAX-OF` using argument records.

```
: MAX-OF ( n.addr n -- max )
  { var table  const n  -32768 num max }
  n 0 DO
    val table  val max  >
    IF  val table  to max  THEN
    ++ table
  LOOP
  val max
;
```

Figure B.3: A definition of `MAX-OF` using argument records

The argument record sets up three entries, a pointer to pass through the table of integers (`table`), a local constant to hold the current value (`n`) and a local variable to hold the current maximum value (`max`). The local variable is initialised to -32768, the minimum signed 16 bit value.

Within the definition, the argument record parameters are preceded by a "*method selection prefix*". The phrase '`val table`' returns the value of the current table item and the phrase '`++ table`' will increment the address referenced by `table` in order to access the next item in the table. The phrase '`to max`' will store the top of the stack in the local variable `max`.

As a final example, figure B.4 shows a string matching routine. It is a simple task to add new parameter types. The "method selection" mechanism is based on object oriented programming techniques[1], thus the same selectors can be used in a polymorphic manner. It is just as simple to add new method selectors as it is to add new parameter types. See (Stoddart 1984) or (Stoddart 1990a) for a full description of argument records.

[1]The "method selector" can be viewed as a *message*, while the "parameter type" is the object *class*. Thus the phrase '`cvar $1`' can be interpreted as instigating an instance `$1` of class `cvar`, while the phrase '`val $1`' can be seen as passing the message `val` to the object `$1`.

```
: $MATCH ( c.addr1 c.addr2 count -- flag )
   { cvar $1  cvar $2  const n }
  TRUE ( if the strings match this will be left )
   n 0 DO
     val $1  val $2  <>
    IF  NOT ( switch flag to false )  LEAVE  THEN
     ++ $1  ++ $2
   LOOP ;
```

Figure B.4: A string match function, using argument records.

B.4 The Multi-Processor FORTH Interpreter

B.4.1 The Users View

The user interface is designed so that a user can choose to interact with any of the processors in the system. It is also possible to define a single command to load and run an application that involves all of the processors.

We refer to the IBM PC as the *host* system, since it provides terminal and mass storage facilities for the *MACH1* which is referred to as the *target* system.

Interaction

At cold start, the user is interacting with the host system. The command `T` switches interaction to the *MACH1*. Where there is more than one *MACH1* board, the commands `T0`, `T1` etc. are provided to connect to a particular processor.

To switch interaction from the target board back to the host, the user should press the $\boxed{\text{ALT-H}}$ key. This will cause the host to exit from the terminal emulation program invoked by `T`. The target system has a special command '`HOST`' this causes the host to exit from the terminal emulation program and continue execution from that point. The target also continues its own execution thus allowing a user to invoke tasks on both target and host systems.

We can show how this works with an example. For this system the screen is split into two windows. Interaction with the target takes place in the one window, whilst interaction with the host takes place in the other (larger) window. Whilst connected to the host system a user could enter:

```
T 1000 2000 DUMP
```

This connects the user to the target by running the terminal emulation utility `T`. The rest of the command line is left to be interpreted when the `T` utility terminates.

Whilst connected to the target the user could enter:

```
HOST 0 1000 DUMP
```

The `HOST` command causes the host system to terminate its execution of `T` and to continue execution by interpreting "`1000 2000 DUMP`". Meanwhile, the target continues execution by interpreting the text that follows `HOST`, the "`0 1000 DUMP`". The target's output goes to the first window, the host's to the second, both host and target will be dumping 1000 bytes of memory to their respective windows.

Messaging

Sometimes it is useful for a word defined on the host to post a message to be interpreted by the target. This is achieved with the word `T"` which has a similar syntax to `."` but which queues the following text string in a buffer which will be interpreted by the target when the `T` command is next invoked.

Suppose we have an application that requires screen 10 to be loaded on both host and target systems, screens 11 to 20 to be loaded by the host and screens 21 to 30 by the target. Now suppose that the target system is to be set running by the command `GO-TARGET` (assumed to be defined a part of the target application code) and the host is to be set running with `GO-HOST` (again assumed to be defined as part of the hosts application code). This can be achieved with the following definition:

```
: RUN
  10 LOAD                \ Load common screen
  11 20 THRU             \ Load Host specific application code
  T"                     \ Target will do:
    10 LOAD                   \ Load common screen
    21 30 THRU                \ Load Target specific application code
    HOST                      \ Release host
    GO-TARGET"                \ Start target application code
  T                     \ Connect with target
  EVAL" GO-HOST"        \ Start Host application code
;
```

Note that `EVAL" GO-HOST"` simply interprets the text string `GO-HOST`. This is necessary because we are assuming that the word `GO-HOST` is not defined when `RUN` is compiled, but is defined by the host specific application code on screen 11 to 20.

Viewing code

A final important feature of the system is the implementation of a `VIEW` facility for both IBM PC and RTX systems. The phrase `VIEW` *<word>* is used to enter the editor in *browse* mode at the screen where *<word>* is defined. The editor also has a *search* facility to locate text strings. Together with `VIEW` this provides a powerful method of reviewing the definition and subsequent usage of FORTH words.

B.4.2 Implementation Notes

Communication between the IBM PC and *MACH1* systems takes place via the dual port memory area. This contains the *MACH1*'s disc buffers and a number of shared data structures.

The *MACH1* sees its keyboard and screen as two "mailboxes" in the dual ported memory. Each mailbox consists of two cells, one of which holds the character in transit, the other providing synchronisation by holding a "*mailbox character available*" flag. The *MACH1* side of the dual-port interface is therefore very simple, all the sophistication is on the IBM PC side.

The IBM PC word `T` actually performs several tasks. It will accept keyboard input and places the key codes into a transfer queue. It also monitors the output mailbox displaying any characters that appear there. Finally, it monitors a dual port data structure through which the *MACH1* posts requests for occasional services. These include the "*save current system*" request, the request to `VIEW` the source code screen for a given definition and the "*continue execution*" request posted by the `HOST` word.

A second host actor transfers characters from the character queue to the keyboard mailbox. Another monitors and acts on mass storage requests and another monitors and acts on requests to access the IBM PC's memory[2].

The system, as described, operates in whichever window `OPERATOR` is currently associated. However, it is easy to provide a dual window system in which interaction with the target takes place in a separate window. To achieve this, another mailbox with associated access functions must be set up with another host actor to display the output of the new mailbox in the target window. Having provided this the *MACH1*'s `EMIT` routine should then be vectored to put the outgoing characters into the new mailbox.

[2]This is needed when the name and string segments are being stored in the IBM PC's memory rather than on the board.

B.5 Code Optimisation

The *RTX-2000* family provides many single op-codes which can replace sequences of two or more standard FORTH operations (Harris Semiconductor 1988). For investigation purposes, we implemented an optimiser which can recognise every "many to one" code reduction sequence, including those which include high level calls and those defined by the user. It uses a tree traversal algorithm and users can add new branches to the tree to include new op-code sequences for optimisation. The algorithm will also change past optimisations if it finds a better one ahead.

This optimiser is supplied but is not built into the system because the implementation algorithm requires more space that is likely to be save by optimisation! We have found that a fairly simple optimiser can achieve 90% of the code space and execution time saved by the full optimiser. This mini-optimiser is permanently loaded and operational by default. Since installing the mini-optimiser, we have not observed any performance degradation.

Theoretically however, an optimiser can interfere with the correct compilation of FORTH-83 Standard code. For example consider the phrase:

```
COMPILE SWAP -
```

If optimisation is on, the sequence '`SWAP -`' will be optimised and compiled into the single op-code `SWAP-`. When the compiled code is executed, this is the op-code that will be compiled by `COMPILE` and not the required `SWAP`. To deal with this, the commands `OPT` and `-OPT` are provided to turn optimisation on and off.

One of the most effective RTX optimisations is the ability to perform a return instruction in parallel with the last op-code of the routine. Most RTX instructions have a return bit, which is set to cause a return to be executed in parallel with the instruction. Normally the compiler checks whether the last instruction in a definition is an RTX op-code primitive and sets the return bit of the op-code if it is. As an example of where this optimisation is inappropriate consider the definition:

```
: ABS  DUP 0< IF  NEGATE  THEN ;
```

When compilation reaches the semi-colon, the most recently compiled operation is `NEGATE`. However, we need to compile a return op-code rather than set the return bit in the negate op-code. To achieve this the definition of `THEN` includes an operation which informs the system that the last op-code compiled can not be optimised. This is transparent to the user but must be considered if the user is defining his own control structure words[3].

B.6 Graphics

The IBM PC version of FORTH++ can support extensive graphics libraries via its interface to C graphics library routines (see Chapter 3). In many applications, the speed of the RTX can be a great help in calculating the form of graphics images. For example, the problem of transforming one graphics image into another by producing a series of intermediate images. Another common example is the generation of fractal images.

These applications produce some interesting problems in terms of debugging application code as the screen that is normally used to observe the progress of our FORTH application by means of stack prints, etc., is now given over to display purposes. FORTH++ helps with this in two ways:

1. It supports a utility which splits the display screen into two areas, one of which is used by the FORTH interpreter while the other is used for graphics display.

[3]Due to the way one constructs new control structures in ANSI-FORTH this is no longer a consideration.

2. It supports the ability to output text to closed windows. Suppose we are using the dual window system described in section B.4.1, in which the target and host interaction take place in separate windows. On entering graphics mode the windows are closed but console output is not vectored to the graphics screen. Therefore, any console output produced during graphics mode will be displayed on exit and the windows are opened again[4].

With the use of network communications via an add on module (see Chapter 2) it is possible to provide a programming environment, where all text output from the "Graphics" system is passed over the network to a "text" system. Thus allowing the programmer to have the normal programming environment on one system whilst displaying the graphics screen on the other system. We have programmed such a system.

B.7 References

ANSI (1991). *ANS ACS X3/X3J14 Programming languages —* FORTH*: Draft Standard* (first ed.). American National Standards Institute.

Harris Semiconductor (1988). *Harris RTX-2000 Programmer's Reference Manual.* Melbourne, FL: Harris Corporation.

Marriot, R. (1989a). *MACH1 Hardware Reference Manual.* Cranleigh, UK: Micro-Amps Ltd.

Marriot, R. (1989b). *MACH2 Hardware Reference Manual.* Cranleigh, UK: Micro-Amps Ltd.

Stoddart, W. J. (1984). Readable and efficient parameter access via argument record. *Journal of* FORTH *Application and Research 3*(1), 61–82.

Stoddart, W. J. (1990a). FORTH++ *Course Notes — Part 3: Argument Records.* Middlesbrough, UK: Teesside Polytechnic.

Stoddart, W. J. (1990b). *MACH1* FORTH++ *evaluation system.* Middlesbrough, UK: Teesside Polytechnic.

Stoddart, W. J. (1990c). *RTX* FORTH++ *MACH1 Specifics.* Middlesbrough, UK: Teesside Polytechnic.

Stoddart, W. J. (1990d). *RTX* FORTH++ *Manual.* Middlesbrough, UK: Teesside Polytechnic.

[4]Due to hardware limitations the contents of the screen are destroyed when entering or exiting graphics mode, thus you must close windows before entering graphics mode and reopen them on exit.