Chapter 1

Overview

This project has the aim of developing software tools and formal notations that facilitate target software development for Multi-processor systems.

As Computer Solutions (our industrial partners) specialise in "Real-Time" and safety critical systems, the work has been slanted toward providing development methods for this form of programming. Computer Solutions are also one of the few companies that specialise in using the FORTH programming language. Thus our main objective is to investigate software development on multi-processor RISC systems under the FORTH development environment.

In this chapter we provide an overview of the entire project, from start to finish. This chapter is intended to introduce the reader to the work, which is described in more detail in the rest of this report. It is also intended to acquaint the reader with the results of this work.

1.1 Introduction

The initial areas of investigation were as follows:

- The provision of a method of communicating between multiple processors, and the message passing system that this would involve.
- The interfaces between FORTH and other high-level languages, for example allowing the developer the freedom to interface with supplier propriety code. This is to be done in such a way that FORTH's interactive user interface is maintained.
- The interface between FORTH and a local area network as a method of providing a multi-processor message passing system.
- The provision of such system on RISC based micro-controllers such as the Novix NC4000, Harris RTX-2000 or other such processors.

We were able to address some of these problems directly. We describe a mixed language interface, an interrupt driven network interface etc. However, many aspects of our investigations proved to be dependent on a more thorough theoretical underpinning of the FORTH language. Thus our attention moved to providing such a foundation. This work mainly consisted of:

- FORTH uses a typeless parameter stack to pass arguments, a programmer must concern himself with the intellectual burden of managing the parameter stack. He must not only know the location

[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.**

of each argument on the stack, but also its logical type. The mismatching of types can be the cause of a subtle logic error. We therefore investigated the possibility of developing a "type algebra" that would allow us to check the type of stack elements. We have suggested how this algebra could be built into a stand alone program, or more properly into the standard compiler mechanism.

- In order to support the specification and use of multi-tasking, we supply a theory of concurrent tasks based on state machines that synchronise on events. This should provide a close match between the theoretical *state machine* and a *task* in the final implementation. To allow people not familiar with formal notations to use this theory we have provided a graphical representation for use with this theory.

- We also looked at how formalisms might be used to define a semantic model of the FORTH language. We have provided a formal base from which the semantic meaning of a program can be derived. We used this formal base to investigate the relationship between the stack based virtual machine and register based target processors.

1.2 Equipment

Novix–4000 Boards: We were using *PC4000* boards made by Silicon Composers. This was a board that contained a Novix NC4016 CPU running at 4 MHz. The board is fitted with an edge connector suitable for connection to an IBM PC.

The *PC4000* board has 512 KBytes of processor memory dedicated to the Novix processor. Some 16 KBytes of this memory is shared with the IBM PC. This "Dual ported memory" provides a mechanism by which the IBM PC can communicate with the Novix.

As the *PC4000* board does not contain any ROM the Novix is placed in an *idle* state when first powered up. It is the responsibility of the IBM PC to load a Novix "boot" program into the dual ported memory and then to provide an interrupt signal that sets the Novix executing the code placed in the dual ported memory.

This memory is located at memory location `00000` on the Novix board. It is mapped into the IBM PC's memory space at a variable location (set via the shorting of jumper switches on the *PC4000* board). The first board had the dual ported memory mapped into the segment address `CC00`, with the second board mapped to segment address `D000`.

RTX–2000 Board: The *MACH1* board, supplied by MicroAMPS Ltd., shares the advantages of prior RTX boards which are plug-compatible with a full IBM PC expansion slot. It also dedicates about 13 square inches of board area (approximately $\frac{1}{3}$ of its full length) to a hardware prototyping area which is suitable for sophisticated project development. An uncommitted backplane connector permits the use of a DB-25 or similar connectors 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 KBytes to 128 KBytes of inexpensive 1- or 0-wait-state SRAM (static random-access memory). The minimum 8 MHz RTX can deliver bursts of 50 MIPS (Million Instructions Per Second) 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. A version of the FORTH++ system was designed to be used with the board and is now distributed with the board as part of a development package.

1.3 FORTH

FORTH is an extensible language based on an abstract processor with two stacks (parameter and return). The FORTH interpreter can be seen as a full macro-assembler and fully integrated operating system for this abstract processor. Due to the size and speed of this abstract processor FORTH is a suitable language for use in "Real-Time" embedded control applications.

It provides us with an interactive debugging environment where we can add new macros (high-level definitions) and new instructions (low-level definitions). It even allows us to extend the macro system by defining new data types (defining words). As this interpreter can also act as a fully integrated operating system, the programmer needs only to learn one set of rules.

However, FORTH is not simply a programming language well suited for embedded applications. It embodies a philosophy of solving problems that is appreciated by the engineers that use it. It has four primitive virtues: (a) Intimacy, (b) Immediacy, (c) Extensibility and (d) Economy. It has two derived virtues: Total Comprehension and Symbiosis.

While FORTH has advantages over many other languages in the efficiency of the code produced and in the development time. More emphasis has been placed on the efficiency of the language than that of the programmer in its evolution thus far (Duff and Iverson 1984).

Duff (1986) claims that FORTH is seriously limited by its lack of sophisticated, structured and consistent data definition facilities. FORTH does not support nested or composite structures and can only associate a single behavior with a data structure. Duff goes on to suggest that object-oriented techniques could provide a model for a more advanced data structuring facility within FORTH.

Standard FORTH is also lacking in a number of other ways when compared to other, similar, languages. Carr and Kessler (1986) identify four major areas where FORTH is lacking:

Symbols: Symbols are distinct from variables. For example given the statement X, when X is interpreted as a variable, the statement would evaluate X and return its value as the result. When interpreted as a symbol the statement simply returns the symbol X.

In FORTH it would be possible to declare a named constant and then use this as a symbolic name throughout the program, but it is the programmer's responsibility to make sure that no two distinct symbols have the same value. Further, if it is possible for the user of the program to input any arbitrary symbol, the programmer will have to explicitly make provisions for this.

Lists: Any kind of structured data can be represented by a list structure (including lists of lists). FORTH has many examples of lists, arguments to a definition, the stacks and wordlists. However, these are all built into the system and are required for the abstract machine. There is currently no standard mechanism for a programmer to maintain his own application based list. There have been a number of implementations of list based data structures, but no mechanism is included as part of the standard.

Automatic Dereferencing: When referencing a variable one is required to dereference it. Ie, one does not simply refer to the variable `X` as this returns the address of the variable. To refer to its value one is forced to dereference the pointer by writing `X @`. This not only clutters up the code but requires conscious thought. It is also a source of many obscure programming bugs when left out.

The majority of languages will automatically dereference a variable for you. You are required to instruct the compiler not to dereference the variable if you wish to refer to it as a pointer. Ie, in C one would write '`X`' to refer to the contents of the variable X and '`* X`' to refer to the address holding the value of X.

Named Parameters: As FORTH does not have named parameters to a definition, they are passed on the stack, a programmer must concern himself with the intellectual burden of managing the parameter stack. By providing a means of associating symbolic names with parameters we release the programmer from this unnecessary detail by automatically handling the parameter stack.

Duff argues that it is not simply that these facilities are missing from FORTH but that they are missing from the language standard. It would not be too difficult to extend a given implementation to provide these facilities, but they must be present in the standard before they can be relied on by an application developer.

For a full introduction to the FORTH language see (Moore 1974; Moore 1980; Brodie 1982; Kogge 1982; Brodie 1984; Stephens and Rodriguez 1986; Rather 1987; ANSI 1991).

1.4 The Novix NC4000 FORTH Engine

In the early 1980's Charles Moore introduced the idea of designing a microprocessor chip around the FORTH language. Later the Novix company produced a RISC based micro-controller that executed a version of FORTH as its native language.

The Novix design takes only 4000 gates in a programmable logic array to implement. The chip has four different data paths and it is possible to use all four within one machine instruction, thereby providing the ability to execute up to five FORTH instructions in one machine cycle. Since the majority of machine instructions are executed in one clock cycle, a system clocked at 10 MHz has a peak effective throughput of 50 MHz or 50 MIPS.

Novix produced this chip and marketed it under the name of "NC4000" (Novix Controller–4000), although the name was later changed to NC4016 to show that it had a 16 Bit data path. The NC4000 was a "first-pass" piece of silicon, and as such it had some hardware problems. A revised version of the chip was designed (the NC6000) but never produced.

Our industrial partners were interested in any programming techniques that would improve the efficiency of their programmers in general and of programming for Novix chip in particular. They foresaw several projects where the use of a cluster of communicating Novix chips would be of interest.

We were given two NC4000 IBM PC plug-in boards, the *PC4000* from Silicon Composers, complete with two different development environments. We have developed a system where a programmer is able to program both Novix systems in a way that they can operate independently and in parallel, communicate with each other and a host system (Appendix A).

1.5 The Harris RTX-2000 FORTH Engine

For various reasons the Novix company had problems which prevented them from producing the new NC6000 chip, or any more NC4000 chips. As one of the chip designers comments (`comp.lang.forth` 1992):

> Novix made the NC4016. It was a first-pass piece of silicon, and had bugs. For various reasons they didn't get any more chips made, thus the supply of Novix chips ran out. Novix licensed the NC6000 design to Harris.
>
> Harris cleaned up the NC6000 design and made it into the RTX processor core. Harris later acquired ownership of the Novix patent application and designs (at first it was just a license agreement).

The Harris corporation took the NC6000 as a basic design and extended it to produce the *RTX-2000* chip set. The comment goes on to say "this was a long, drawn out, and messy affair". As Computer Solutions were acting as agents for Novix in the UK they found themselves without a product, a rival company having already arranged a supply deal with Harris for the *RTX-2000* systems.

This turn of events had a major effect on the work that we were doing. Rather than concentrating on a particular hardware architecture, we focused more of our attention on the programming environment.

To the extent that we were still interested in the hardware architecture, we switched our attention to the Harris chip. The development work that had been performed up to that stage was now obsolete. In order to continue with our development work, we obtained a *RTX-2000* based system. However,

work with this could not form the main focus of our research as the machine was of little interest to our industrial partners.

The reasons why we abandoned our interests in the Novix were outlined by one of the designers (`comp.lang.forth` 1992):

> The reason you had to switch from Novix to Harris was because Novix:
>
> 1. Didn't come through with fixed versions of its chip.
> 2. Had difficulties that prevented them from acquiring more chips of even the first design.
> 3. Had sold the design to Harris, who became the primary supplier.

As we were still interested in the hardware, we purchased a *PP2000* board, an *RTX-2000* based IBM PC plug in board from SMIS (Surrey Medical Imaging Systems Ltd.). The development software supplied with the system was a very basic system with little (almost no) host services. We later developed a version of our FORTH system for the processor (Appendix B).

1.6 Networks

Our industrial partners wanted the ability to have several IBM PC's around a workshop, all linked to the one "Master" system, thus giving engineers the ability to interrogate a section of the plant that the IBM PC in question is unable to monitor. Any system we develop would not only have to operate correctly with multiple processes on the same processor but also with multiple processes on multiple processors connected via some kind of Local Area Network (LAN).

It should be possible to link several processes together, even though they are not physically located on the same system. By use of a LAN it should be possible for one process to send a message to another process without knowing where the other process is physically located. Where a process is physical located (on a different or the same machine) is of no interest to the process. It is the responsibility of the message passing system to hide this information from the process.

We investigated the IBM NETBIOS system as a standard for (a generalised method of) interfacing with LANs. The NETBIOS system is designed to operate in parallel with any application. We have developed a FORTH interface that exploits this ability (Chapter 2). We have also developed several demonstration programs to show how this system can be used to pass messages from one IBM PC to another, one of which is discussed along with the interface.

1.7 Mixed Languages Interface

There are several FORTH systems available that allow the programmer to invoke functions written in C. However these systems are either written in C, or are designed with one particular compiler in mind, thus are "tied" to a given compiler.

Our mixed languages interface is an interface between the FORTH programming environment and any other programming language. The system (described in Chapter 3 and Appendix C) is designed to interface to code written using Microsoft C, however, the system is language independent, thus allowing the developer to interface with code written in any language (or second party supplied code). Although the system was required for the Microsoft C compiler, it was developed with the Turbo C compiler from Borland. We have tested the "portability" of this system, by compiling the same code under the Microsoft C, Turbo C, ZorTech C and C++ compilers.

1.8 Formal Methods

Formal notations provide a way of specifying a problem in a precise mathematical notation. The specification is an abstract mathematical model of the problem, describing what the system has to

do, rather than how it is to be done. The notations use set theory and first-order predicate logic to build such models. There are several reasons for using formal notations:

Understanding: A formal specification can be passed from one person to another without the possibility of misunderstanding. Due to the ambiguity of natural language, we can never be certain if another person fully understands our meaning from a statement. When using formal notations we can be sure that our exact meaning is presented, as we are using a precise mathematical language (Spivey 1989; McMorran and Nicholls 1989).

Manageability: It has been found that, when specifying large systems, the formal specification is a great deal smaller than the natural language specification. Additionally, the writing of the formal specification can bring out some of the more complex problems that would have remained hidden in the natural language specification (Hayes 1987; Nash 1989; Phillips 1989).

Reasoning: As the specification has been written using a formal notation that is firmly grounded in mathematics, we can mathematically reason about the specification (Woodcock and Loomes 1988; Morgan 1990).

Requirement: There are several companies that now insist on the use of formal methods on safety critical systems. The British Ministry of Defence require formal methods to be used on high level safety critical systems. Lloyds Register now advises that a formal model of all new safety critical systems should be presented before insurance cover is issued.

1.8.1 Formal Forth

As the FORTH language is predominantly used in the fields of real-time process control and safety critical systems, it will become necessary to have the ability to prove that a program meets its specification in every detail.

As the use of formal methods becomes more widespread, it is becoming more important that systems are developed from a formal specification. Lloyds Register now recommend that highly safety critical systems are formally modelled before implementation. The British Ministry of Defence now insist that all "safety critical" software is formally specified.

Due to the philosophies behind FORTH and the nature of the FORTH abstract machine, it is possible to provide a set of tools that will aid a designer/programmer in ascertaining whether a FORTH program meets its (formal) specification. We have performed preliminary work that has laid down the foundations of such a system (Chapter 4). We have provided a mathematical toolset that will allow designers to produce mathematical models of their actual program thus allowing them to conduct proofs on the program and to compare them against proofs conducted on the specification.

In this toolset, we treat the stacks as a sequence of untyped elements. This has lead to the development of a compiler optimisation technique that uses this idea (Chapter 5). It keeps the top three elements of the stack in internal registers (this is only applicable to systems with large register files). Unlike traditional system that keep the items in specific registers, our system allocates the internal registers *dynamically*. We use a *stack image* to keep track of which stack element is in which register, thus it is possible to obtain 100% optimisation on certain stack manipulation operations.

1.8.2 Type Algebra

One of the limitations of programming in FORTH is also one of its major advantages. We refer to the use of a parameter stack for the holding and passing of parameters. The use of a parameter stack allows us to write re-entrant code very simply. It means that the programmer must keep an image of the stack in his mind whilst programming. As the stack is effectively type-less it allows him to "cast" data items without recourse to inefficient subroutines (as in C). This ability is one used by most FORTH programmers and can assist in keeping program development time down.

This ability also carries with it a problem when the programmer is unable to keep track of all of the items on the stack. As the system effectively performs all casting for him, it is possible for him to cast an item that is not required at that point. For example, to convert an *integer* into an *execution-token*.

Jaanus Pöial, of Tartu University, has developed a "Stack Type Algebra" that provides the capability of checking for such unintended type mismatches (Pöial 1990). This system works when all stack items are of a known type. We have developed a similar algebra based around his ideas. Our type algebra (presented in Chapter 6) includes the capability of handling items of variable type, in addition to catering for program structures and conditional execution. We have used the rules from this algebra to specify the action of a "type checker" program (Chapter 7) that should be able to scan a FORTH program and state whether it is "type correct".

1.8.3 The Event Calculus

The "Event Calculus" is a diagrammatic notation which provides an easily used means of formally specifying the behaviour of concurrent systems (Chapter 8). It can describe synchronous and asynchronous communications, data flow modelling and function application, and the expression of temporal constraints. It also has the ability to abbreviate the description of complex state changes, such as data base updates via the use of Z schemas. See (Woodcock and Loomes 1988; Spivey 1989; Diller 1990) for an introduction to the Z notation.

Due to the diagrammatic nature of the calculus, it appears relatively easy for a non specialist to use when compared to other (event based) process algebras such as CSP (Hoare 1985), CCS (Milner 1989) and LOTOS (Brinksma and Bolognesi 1987). As the calculus is also based on formal notations, it gives good control of levels of abstraction that can be used in a model. As a specification produced using the calculus has an underlying formal specification, it is possible to use this specification for deriving proofs of the system begin modelled.

1.9 References

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

Brinksma, E. and T. Bolognesi (1987). Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems 14*, 25–59.

Brodie, L. (1982). *Starting* FORTH (second ed.). London: Prentice Hall International.

Brodie, L. (1984). *Thinking* FORTH. London: Prentice Hall International.

Carr, H. and R. R. Kessler (1986). FORTH and AI? *Journal of* FORTH *Application and Research 4*(2), 177–180.

`comp.lang.forth` (1987–1992). Usenet newsgroup.

Diller, A. (1990). *Z: An introduction to Formal Methods*. London: John Wiley & Son.

Duff, C. B. (1986). ACTOR, a threaded object-oriented language. *Journal of* FORTH *Application and Research 4*(2), 155–161.

Duff, C. B. and N. D. Iverson (1984). FORTH meets SmallTalk. *Journal of* FORTH *Application and Research 2*(3), 7–26.

Hayes, I. (Ed.) (1987). *Specification Case Studies*. Computer Science. London: Prentice Hall International.

Hoare, C. A. R. (1985). *Communicating Sequential Processes.* Computer Science. London: Prentice Hall International.

Kogge, P. M. (1982, March). An architectural trail to threaded code systems. *IEEE Computer*, 22–32.

McMorran, M. A. and J. E. Nicholls (1989, July). Z user manual. Technical Report TR12.274, IBM Hursley Park. Version 1.0.

Milner, R. (1989). *Communication and Concurrency.* Computer Science. London: Prentice Hall International.

Moore, C. (1974). FORTH: a new way to program a mini-computer. *Astronomy & Astrophysics Supplement 15*, 497–511.

Moore, C. (1980, August). The evolution of FORTH: an unusual language. *Byte 5*(8), 76–92.

Morgan, C. (1990). *Programming from Specifications.* Computer Science. London: Prentice Hall International.

Nash, T. (1989). Using Z to describe really large system. In *Proc. Z Users Meeting*, pp. 150–178. Oxford.

Phillips, M. (1989). CICS/ESA 3.1 experiences. In *Proc. Z Users Meeting*, pp. 179–185. Oxford.

Pöial, J. (1990). The algebraic specification of stack effects for FORTH programs. In *Proc. FORML Conf., Proc. EuroFORML Conf.*, San Carlos, CA. FORTH Interest Group.

Rather, E. D. (1987). FORTH programming language. *Encyclopedia of Physical Science & Technology 5.*

Spivey, J. M. (1989). *The Z Notation: A Reference Manual.* Computer Science. London: Prentice Hall International.

Stephens, C. L. and R. M. Rodriguez (1986, July). *Poly*FORTH: an electronics engineer's programming tool. *Software Engineering Journal 1*, 154–158.

Woodcock, J. C. P. and M. Loomes (1988). *Software Engineering Mathematics: Formal Methods Demystified.* London: Pitman Publishing Ltd.