Practical and Theoretical Aspects of Forth Software Development

Peter J. Knaggs

A thesis submitted in partial fulfillment of the requirements of the *University of* Teesside for the degree of Doctor of Philosophy.

The *University of* Teesside in collaboration with Computer Solutions Limited

March 1993

Practical and Theoretical Aspects of Forth Software Development

March 1993

University of
TEESSIDE

Practical and Theoretical Aspects of Forth Software Development

Abstract

This is an investigation into the use of the FORTH programming environment. The main areas of enquiry were: interfacing FORTH to other languages; interfacing FORTH and local area networks; and the use of RISC processors with stack based architecture such as the NC4000 and Harris RTX series.

We describe how to interface FORTH and C. We also provide a system with a multi-tasking interrupt driven interface to the IBM NETBIOS networking software and a simple, generic, method of task activation through message passing.

Many aspects of the investigation proved to be dependent on a more thorough theoretical underpinning for the FORTH language. The use of a typeless parameter stack means that a programmer must concern himself with the intellectual burden of managing the parameter stack. The mismatching of stack elements can be the cause of subtle logic errors. We therefore investigated the possibility of developing a "type algebra" that would allow us to develop a typed version of FORTH. This thesis includes a theory for a "type signature algebra" for the stack based argument passing method used by FORTH.

To support the use of multi-tasking we provide a simple, but formal, theory of concurrent tasks based on state machines that synchronise on events. This has a graphical notation for people who are not familiar with formal notations.

We also looked at how formalisms might be used to define a semantic model for the FORTH language and how formalisms can help to define the relationship between FORTH's stack based virtual machine and register based target processors.

Contents

1 Overview **1**
1.1 Introduction . . . 1
1.2 Equipment . . . 2
1.3 FORTH . . . 2
1.4 The Novix NC4000 FORTH Engine . . . 4
1.5 The Harris RTX-2000 FORTH Engine . . . 4
1.6 Networks . . . 5
1.7 Mixed Languages Interface . . . 5
1.8 Formal Methods . . . 5
1.8.1 Formal Forth . . . 6
1.8.2 Type Algebra . . . 6
1.8.3 The Event Calculus . . . 7

2 Using IBM's NETBIOS **8**
2.1 Introduction . . . 8
2.2 Functions . . . 8
2.2.1 Naming . . . 8
2.2.2 Sessions . . . 9
2.2.3 Datagrams . . . 9
2.2.4 Broadcasting . . . 9
2.2.5 House keeping . . . 9
2.3 Invoking NETBIOS Functions . . . 9
2.4 Multi-Tasking . . . 10
2.5 Examples . . . 10
2.5.1 Block Transfer . . . 10
2.5.2 Net-Chat . . . 11
2.6 Problems . . . 11
2.6.1 *Poly*FORTH . . . 11
2.6.2 Interrupts . . . 11
2.6.3 Porting . . . 12
2.7 Comparison with C interface . . . 12
2.8 Interface Code . . . 12
2.8.1 Error Handler . . . 12
2.8.2 Network Control Block . . . 14
2.8.3 Assembler Interface . . . 14
2.8.4 Low-Level interface . . . 16
2.8.5 General Support . . . 17
2.8.6 Naming Support . . . 18
2.8.7 Session Support . . . 18
2.8.8 Datagram Support . . . 20

2.8.9 Broadcast Support . . . 20
2.9 The "Net-Chat" Application . . . 21
2.9.1 Memory Buffers . . . 21
2.9.2 Listening . . . 22
2.9.3 Sending . . . 23
2.9.4 Initialisation . . . 23
2.9.5 Close Down . . . 25

3 Mixed Languages interface **26**
3.1 Principles . . . 26
3.2 Argument Passing . . . 26
3.3 Programming . . . 26
3.4 The C Heap . . . 27
3.5 Organisation . . . 28
3.6 Generalisation . . . 28

4 Formal FORTH **29**
4.1 Introduction . . . 29
4.2 The FORTH Toolbox . . . 29
4.3 The Basic Model . . . 30
4.4 Word Definitions . . . 30
4.5 Immediate Words . . . 30
4.6 Storage Units . . . 30
4.7 Stacks . . . 31
4.8 Code Definitions . . . 31
4.9 Wordlists . . . 32
4.10 Defining words . . . 32
4.10.1 High-Level words . . . 33
4.10.2 Immediate words . . . 33
4.10.3 Code words . . . 33
4.11 Dictionary Searching . . . 34

5 Stack Optimisation **35**
5.1 Introduction . . . 35
5.2 Code Generation . . . 37
5.3 Inline Compilation . . . 37
5.4 Peep-Hole Optimisation . . . 38
5.4.1 Conditionals . . . 39
5.5 Registers . . . 39
5.6 Optimisation using a Stack image . . . 40
5.6.1 Argument Passing . . . 40
5.6.2 Conditional execution . . . 42
5.6.3 Looping structures . . . 43

6 The Cell Type **44**
6.1 Introduction . . . 44
6.2 Stack Types . . . 44
6.3 Notation . . . 45
6.4 Rules . . . 45
6.4.1 Composition Rules . . . 45
6.4.2 Reduction Rules . . . 46
6.4.3 Wildcard Rules . . . 46

6.5 Simple Examples . . . 47
6.6 Multiple Signatures . . . 48
6.7 Pass by reference . . . 48
6.8 Control Structures . . . 49
6.9 Casting . . . 50
6.10 Strong vs Weak Typing . . . 50
6.10.1 Strong Typing . . . 50
6.10.2 Weak Typing . . . 51

7 A FORTH Type Checker **52**
7.1 Invocation . . . 52
7.2 Stack Notation . . . 52
7.3 Commands . . . 53
7.3.1 Classes . . . 53
7.3.2 Type Command . . . 54
7.3.3 Stack Command . . . 55
7.3.4 Assume Command . . . 55
7.3.5 Assert Command . . . 56
7.3.6 Syntax Command . . . 56
7.4 Variable Stack Items . . . 56
7.4.1 Or — | . . . 56
7.4.2 Alternative descriptions — + . . . 57
7.5 Flow Control . . . 58
7.6 Defining words . . . 59
7.6.1 Pre-defined . . . 59
7.6.2 User-defined . . . 59
7.7 Vocabularies . . . 60
7.8 Error Log . . . 60
7.8.1 Error report . . . 60
7.8.2 Verbose reports . . . 60
7.8.3 Statistics . . . 61
7.8.4 Statistics flag . . . 61
7.9 Problems . . . 61

8 The Event Calculus **63**
8.1 Introduction . . . 63
8.2 State Machines . . . 64
8.3 The Formal Model . . . 65
8.4 An Algebra of machine behaviours . . . 67
8.5 Labelled Transitions . . . 67
8.6 Simple Examples . . . 68
8.6.1 The specification of mutual exclusion (without fairness) . . . 68
8.6.2 Asynchronous Events . . . 69
8.6.3 Value passing . . . 71
8.6.4 Mutual Exclusion with fairness . . . 71
8.7 A GCD algorithm, modelling parameter passing and procedure call . . . 72
8.8 Variables and Scopes . . . 73
8.9 Time . . . 74
8.10 The Dynamic model . . . 76
8.11 Combining the Event Calculus with Z schema calculus . . . 78
8.12 A Distributed seat booking system . . . 80

9 Conclusions and Recommendations **84**
9.1 Introduction . . . 84
9.2 Networks . . . 85
9.3 Mixed Languages Interface . . . 85
9.4 Formal FORTH . . . 85
9.5 Stack Optimisation . . . 86
9.6 Type Algebra . . . 86
9.7 FORTH Type Checker . . . 86
9.8 The Event Calculus . . . 87
9.9 Future Directions . . . 87
9.9.1 Type Algebra . . . 87
9.9.2 Formal FORTH . . . 87
9.9.3 Event Calculus . . . 88

Bibliography **89**

Bibliography **89**

A Communicating Novix NC4016s **95**
A.1 Introduction . . . 95
A.2 Programming . . . 95
A.2.1 cmFORTH . . . 95
A.2.2 *SC*FORTH . . . 95
A.2.3 *Poly*FORTH . . . 96
A.2.4 FATWIN . . . 96
A.3 Single Boards . . . 96
A.4 Host Services . . . 97
A.5 Parallel Boards . . . 97
A.5.1 First Method . . . 98
A.5.2 Second Method . . . 99
A.5.3 Comparison . . . 99
A.6 Communicating Systems . . . 100
A.6.1 Hardware Restrictions . . . 100
A.6.2 Communication . . . 100
A.7 Code . . . 101
A.7.1 Multiple Boards "Boot" code . . . 102
A.7.2 First attempt at providing Host Services . . . 103
A.7.3 Revised Boot and Host code . . . 104

B FORTH++ and the MACH1 **107**
B.1 The *MACH1* . . . 107
B.2 The *MACH2* . . . 107
B.3 FORTH++ . . . 107
B.3.1 Memory Organisation . . . 108
B.3.2 Multi-Tasking and Windows . . . 108
B.3.3 Argument Records . . . 108
B.4 The Multi-Processor FORTH Interpreter . . . 110
B.4.1 The Users View . . . 110
B.4.2 Implementation Notes . . . 111
B.5 Code Optimisation . . . 111
B.6 Graphics . . . 112

C Mixed Languages Interface: Source Code **114**
C.1 Loader 114
C.2 Making the loader 126
C.3 Overlay initialisation 126
C.4 Context Switching 128
C.5 Stack access 132
C.6 User code 134
C.7 Making the C Overlay 139

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 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.

Chapter 2

Using IBM's NetBios

A general overview of the Ibm NetBios system is given and its Multi-Tasking abilities are discussed. A Forth interface that exploits these is presented and a "Net-Chat" program, which illustrates the integration of NetBios with Forth's Multi-Tasker is described.

2.1 Introduction

The *Net*work *B*asic *I*nput/*O*utput *S*ystem (NetBios) is an "application program interface" (Ibm Corporation 1987) between an application task and a *L*ocal *A*rea *N*etwork (Lan) designed to provide a common communication capability between Ibm Pcs and compatibles. It has been implemented on a wide variety of physical networks including Ethernet, Token ring, Insertion ring, etc.

NetBios provides a communication link (or connection) between named entities using two main forms of communication, known as *sessions* and *datagrams*. Any application may add a name to the network. In a Forth Multi-Tasking system it would be possible to provide two separate application tasks, each with its associated name, on the same host machine. The two tasks would then communicate with each other using the NetBios and neither task need know where the other is situated.

All requests to the NetBios are made using a *N*etwork *C*ontrol *B*lock (ncb) supplied by the application program. The ncb holds parameters for the network call and, on completion, contains status information.

2.2 Functions

The functions provided by NetBios can be broken down into five groups: Naming, Sessions, Datagrams, Broadcasting and General Housekeeping. (See (Ibm Corporation 1987) or (Nine Tiles 1988b) for a complete breakdown of the NetBios functions.)

2.2.1 Naming

Each network card has its own unique physical name. To use this name in an application would be too restrictive as such an application would be forced to know with which physical system to communicate. NetBios provides some naming capabilities to allow applications to refer to logical, rather than physical, names thus allowing a network application to be independent of any physical machine.

The *Add Name* function will add a unique name to the network. This will provide a logical name for the physical system performing the add name function. Each physical system may have several logical names that could be used by different tasks or applications.

The function *Add Group Name* will add a group name to the network. Several machines may have the same group name and will then be classed as members of the same group. This facilitates communication to a selected group of machines.

Remove is used to remove a name from the Network. If the name is an individual name, the name is completely removed. If it is a group name, the machine is removed from the group.

2.2.2 Sessions

A Session provides a one-to-one connection, analogous to a telephone call.

A Session is started by one application making a call to another. The called application must be listening for an incoming call. To call another application, the *Call* function is used. The *Listen* function is used to wait for an incoming call, while *Hangup* is used to disconnect the call. If you call a group name, only one member of the group will receive the call.

Once the connection has been established, the applications can exchange data (up to 64K at a time) with the guarantee that it will arrive. To exchange data, one application must use the *Transmit* function while the other is using a *Receive* function. If one side issues a transmit before the other has issued the corresponding receive, the data will be buffered until the receive is issued.

2.2.3 Datagrams

A datagram is a one-shot communication of up to 512 bytes.

A *Datagram Transmit* will send a datagram to a given name. The receiving name must be waiting to receive it otherwise it will be lost. When a datagram is sent to an individual name, only that name will receive it. However, if it is sent to a group name, all the members of that group will receive a copy.

A *Datagram Receive* will wait for a datagram to be received by a given name. A datagram transmitted from any name to the given name will be accepted.

2.2.4 Broadcasting

A Broadcast is a special form of datagram that is sent to all names. The *Broadcast Transmit* function will send a datagram to all names known to the network. The *Broadcast Receive* function is similar to the datagram receive function, except it will only receive a Broadcast message.

2.2.5 House keeping

There are four basic functions that are designed for the network manager to control the network system. The *Reset* function is used to totally reset the network card. *Network Status* will return the current status of the network card. A *Cancel* function is used to cancel a given command. Finally the *Un-Link* function disconnects from a remote disk server.

2.3 Invoking NETBIOS Functions

All NETBIOS functions are invoked in the same manner. The data required by the function is placed in the relevant fields of the NCB and the NETBIOS system call is invoked. This will take the NCB and post it into the NETBIOS for processing. The actual processing of the function is interrupt driven and will run concurrently with the application program.

NETBIOS has three different ways of returning back to the application program. The first is referred to as a *Wait* function, where NETBIOS will process the complete function before returning to the application.

The second is to post a *No-Wait* function. NETBIOS will add the function to its internal list of functions and return to the application directly. The application program must poll the "command complete" flag of the NCB to determine if the NETBIOS has completed the function.

The final method is to post a *No-Wait* function giving the address of an interrupt routine. The NETBIOS will add the function request to its internal list and return to the application program. When the function has been completed, it will invoke the given interrupt code.

2.4 Multi-Tasking

In order to exploit the concurrent execution abilities of FORTH and the NETBIOS, we use the "*No-Wait with Interrupt*" invocation method. When a NETBIOS function is used, the invoking task will typically execute a `STOP` after making the NETBIOS call.

In the FORTH/NETBIOS interface, a field has been added to the NCB to store the identity of the invoking task. The interrupt routine passed to the NETBIOS is always a "*wake task*" routine that extracts the task identity from the NCB and sets the task status to active, thus waking the task associated with the NETBIOS function.

More than one task can have a NETBIOS request pending. For example, one task may be waiting on a *Broadcast Receive*, whilst another is waiting on a *Datagram Transmit*. Any one task may have several NETBIOS requests pending. For example, in the "Net-Chat" application, one of the tasks posts four *Datagram Receive* requests to ensure that no incoming datagrams are lost (see sections 2.5.2 and 2.9). When the task is made active it has to poll the NCBs of the pending commands in order to discover which of them has completed.

2.5 Examples

In this section we provide the reader with two examples of how the FORTH/NETBIOS interface can be used.

2.5.1 Block Transfer

To transfer a block of data from one system to another, both systems must make themselves known to the network. This would be done by each of them creating an NCB. They would then add their individual names to the network.

System 1

```
NEWNCB NCB
" PETER" NCB ADD-NAME
```

System 2

```
NEWNCB NCB
" JOHN" NCB ADD-NAME
```

Now `PETER` may call `JOHN`. The connection is made when Peter is calling John and John is listening for a call from Peter (or when John makes a call to Peter, although Peter must be listening for the call in this case).

```
" JOHN" NCB PHONE
STOP
```

```
" PETER" NCB LISTEN
STOP
```

`PETER` will now send a block of data over the network to `JOHN`.

```
9 BLOCK  ( Address of buffer )
1024     ( Number of bytes )
NCB      ( NCB to use )
TX STOP  ( Transmit )
```

```
10 BLOCK  ( Address of buffer )
1024      ( Number of bytes )
NCB       ( NCB to use )
RX STOP   ( Receive )
```

One of the systems must now disconnect. Our convention is that the caller is in charge of the connection and hence is responsible for the disconnection.

```
NCB HANGUP  ( Disconnect )
```

The **STOP**s are required to allow other tasks to continue executing and to synchronise communications.

2.5.2 Net-Chat

A simple example application program has been developed along the lines of the "Net-Chat" program by Glass (1989). This is a Citizen Band radio emulation, in that if anyone sends a message over "Net-Chat", it will be received by all other systems running the application.

The basic principle to a "Net-Chat" implementation is to have a group name of "**NET-CHAT**" and an individual name for each person on the system. The screen is divided into two sections with a small 5 line window provided for the Net-Chat display and a larger second window displaying the normal **OPERATOR** environment.

A task ("**CHAT-TASK**") will post four *Datagram Receive* requests on the group name **NET-CHAT**. When a datagram is sent to **NET-CHAT**, all the members in the group will receive a copy (including the sender). When receiving messages, **CHAT-TASK** will scan through the NCBs to discover which one was honoured. It will take the message buffer of the NCB, display it in the Net-Chat window and will use the NCB to post a new *Datagram Receive* request. If only a single *Datagram Receive* was posted, it would be possible to miss a datagram that arrives between the previous datagram being received and the *Datagram Receive* request being re-posted.

To send a message, the user must type the word **CHAT**. This will ask for a message to be sent. It will send the message buffer to the group name **NET-CHAT**.

The code and a more detailed description, is given in section 2.9.

2.6 Problems

As this system was originally intended for use with the Novix micro-processor system, it was developed using the *Poly*FORTH system. It was later ported to the FORTH++ system (see chapter 1). In this section we describe some of the problems that had to be overcome before this system became fully operational.

2.6.1 *Poly*FORTH

The *Poly*FORTH system operated correctly when used in a network based environment. When we loaded the NETBIOS interface code, the system stopped operating altogether. The *Poly*FORTH code appeared to be correct while the interface code also appeared to be correct.

After some experimentation, we discovered that the problem only occurred when the *Poly*FORTH serial communications package was loaded. By forcing the system *not* to load this package, the problem was overcome. In order to continue with this project, it was necessary to convert this system for use with the FORTH++ system. Thus, the real cause of the problem was never investigated.

2.6.2 Interrupts

The original version of this system used the *No-Wait and Poll* method of posting a NETBIOS function. This meant that when an application task had posted a NETBIOS function, it would enter a loop testing the command complete flag of the relevant NCB. As the task is actively waiting for the function to complete, it is scheduled for time by the multi-tasking scheduler.

The system was redeveloped to take advantage of the "*No-Wait with Interrupt*" ability of the NETBIOS. The system developed to utilise this facility is described in section 2.4. The task posting a NETBIOS function is allowed to continue execution. Eventually the task will execute a **STOP**. When the NETBIOS function has been completed the NETBIOS will invoke the given interrupt code. This code will reset the associated task's status to active thereby making sure that the task will be executed.

This allows a task to post as many NETBIOS functions as it requires. It also allows the task to be removed from the scheduler's active tasks list. When the NETBIOS function has completed[1] it will add the task to the active task list, thus removing the responsibility of polling the command complete flag altogether.

2.6.3 Porting

The port from *Poly*FORTH to FORTH++ was a very simple one with only one small problem. None of the code had to be changed with the exception of the two machine code words.

The *Poly*FORTH assembler system is designed to be as processor independent as possible, while the assembler provided with the FORTH++ system is designed around the Int$_e$l 80x86 family of processors. The two machine code words had to be converted from the *Poly*FORTH assembler form into the FORTH++ form. The function of the code was not altered in any way, nor was the machine code produced altered. The only alteration was to the source code in order to produce the same object code.

We also took this opportunity to exploit FORTH++'s ability of holding 64 KBytes of strings to enhance the error messages and improve the error handling provided by the interface.

2.7 Comparison with C interface

When compiled, the NETBIOS interface shown in section 2.8 forms a run-time library. The library comprises of 186 lines of FORTH code and compiles to just 1.2 KBytes (when compiled under FORTH++). A simple C interface (taken from Schwaderer (1988)) takes some 110 lines of code (1.8 KBytes when compiled) and 270 lines of compile time definitions to provide the same functionality as the `(net)` word. The C interface requires the application developer to have a full knowledge of the NETBIOS and the NCB. A full C library that provides the same functionality as the interface shown in section 2.8 requires some 115 KBytes (when compiled).

As the C language does not directly cater for multi-tasking, such an interface has to use the *No-Wait* or *No-Wait and Poll* techniques for invoking a NETBIOS function. Using the *No-Wait and Pool* technique puts the onus on the application programmer to poll the command complete flag, thus does not provide the full abstraction one might hope for.

2.8 Interface Code

The following is an annotated source listing of the NETBIOS Interface provided for use with the FORTH++ system.

2.8.1 Error Handler

Here we define the word "`(netable)`" to display an understandable network error message. It only displays the errors documented in the NETBIOS manual (IBM Corporation 1987). Any error code not defined in the manual will be displayed as "Unknown".

```
HEX

: (netable)
  CASES
    01 CASE ." Illegal Buffer Length"   END-CASE
    03 CASE ." Illegal Command"         END-CASE
    05 CASE ." Timed Out"               END-CASE
    06 CASE ." Message Incomplete"      END-CASE
    08 CASE ." Illegal Session Number"  END-CASE
```

[1] or any one of the NetBios functions associated with the task has completed.

```
    09 CASE ." No Resource Available"   END-CASE
    0A CASE ." Session Closed"          END-CASE
    0B CASE ." Command Cancelled"       END-CASE
    0D CASE ." Local Duplicate Name"    END-CASE
    0E CASE ." Name Table Full"         END-CASE
    0F CASE ." Name Not Registered"     END-CASE
    11 CASE ." Session Table Full"      END-CASE
    12 CASE ." Call Rejected"           END-CASE
    13 CASE ." Illegal Name Number"     END-CASE
    14 CASE ." Destination Not Found"   END-CASE
    15 CASE ." Name Not Found"          END-CASE
    16 CASE ." Remote Duplicate Name"   END-CASE
    17 CASE ." Name Deleted"            END-CASE
    18 CASE ." Session Aborted"         END-CASE
    21 CASE ." NetBios is busy"         END-CASE
    23 CASE ." Invalid LAN number"      END-CASE
    24 CASE ." Command not found"       END-CASE
    26 CASE ." Illegal Cancel Command"  END-CASE
    34 CASE ." Illegal Data Format"     END-CASE
    DROP
    ." Unknown"
  END-CASES
;
```

We now define the default action to be taken when a network error occurs. This is defined in the word `(neterror)`, it will abort the current operation and display an error message of the form:

```
Network Error code: 15 (Name Not Found)
```

Displaying the network return code and a text message relating to the code (if known). Note that the word `?CASE` takes a flag of the stack and executes the code between the `?CASE` and the `END-CASE` if the flag is true, otherwise it simply skips over the code.

```
: (neterror) ( n -- )
  CR ." Network Error code: " DUP . ASCII ( EMIT
  CASES
                  FF  CASE ." Not Finished"           END-CASE
    DUP 50 FF WITHIN ?CASE ." Hardware Fault"    DROP END-CASE
    DUP 40 50 WITHIN ?CASE ." Unusual Condition" DROP END-CASE
    (netable)
  END-CASES
  ASCII ) EMIT CR ABORT
;

DECIMAL
```

Next we define the network error handling. This is provided by the word `NETERROR`, it takes the NetBios return code and invokes the word, the execution token of which is stored in the user variable `'NETERROR`, if there has been an error, otherwise it simply removes the return code. The defining word `USER*` is used to define a user variable at the next free slot in the user area.

```
USER* 'NETERROR

: NETERROR ( n -- )
  ?DUP IF  'NETERROR  @ EXECUTE  THEN
;
```

Finally we initialise the network error handler to be our default error handler.

```
' (neterror) 'NETERROR !
```

2.8.2 Network Control Block

In this part of the system we define the logical names for the fields of the network control block (NCB), these are the names as given in the manual. It should be noted that we are using the **@** symbol to indicate a segment and offset pair in accordance with the manual. The run-time action of these words is to return the address of the given field in the given NCB.

The word **pos** is a defining word, the size of the field (in bytes) is given on the stack, **pos** will then define a word, the action of which is to add the required byte offset to an address in order to give the address of the required field. We have added the **TASK@** field to hold the address of the invoking task. This is not part of the standard NCB structure but has been added to allow the interrupt routine to identify the associated task. Finally, the constant **ncb_size** is defined to hold the size of our NCB structure (in bytes).

```
: pos CREATE OVER C, + DOES> C@ + ;

0  \ Initial byte count

1 pos CMD        1 pos RETCODE   1 pos LSN         1 pos NUM
4 pos BUFFER@    2 pos LENGTH   16 pos CALLNAME   16 pos NAME
1 pos RTO        1 pos STO       4 pos POST@       1 pos LANA_NUM
1 pos CMD_CPLT  14 pos RESERVED  4 pos TASK@

CONSTANT ncb_size
```

Next we define some NCB control words. The first of these is **NEWNCB**, this will allocate **ncb_size** bytes of memory to act as an NCB. It also creates a word, the action of which is to place the address of this memory area onto the stack.

```
: NEWNCB ( -- )
  CREATE HERE ncb_size DUP ALLOT ERASE
;
```

The second control word is **TIME-OUT**, this is used to set the "Receive" and "Send" time-outs for a given NCB. The time-outs are given in increments of $\frac{1}{2}$ seconds. The system is initialised to no time-outs by default.

```
: TIME-OUT ( Receive-Time-Out Send-Time-Out NCB -- )
  DUP STO ROT SWAP C! RTO C!
;
```

The last of the NCB control words is **COPYNCB**. This is used to copy the data from one NCB to another.

```
: COPYNCB ( Source-NCB Destination-NCB -- )  ncb_size CMOVE ;
```

2.8.3 Assembler Interface

This is where we have developed the assembler code that interfaces between the FORTH++ system and the NETBIOS.

First, we define a word **FIELD** that returns the byte offset of a named field in the NCB. As this word is being defined exclusively for use in code level definitions, we place its definition in the **ASSEMBLER** wordlist.

```
ASSEMBLER DEFINITIONS

: FIELD ' >BODY C@ ;
```

```
FORTH DEFINITIONS
```

We now define the assembler word **(post)**. This is the code that will be invoked by NETBIOS when it has completed a *No-Wait with Interrupt* operation. On entry to this code, the **ES:BX** register pair are pointing to the start of the NCB that has completed.

This code is invoked by an interrupt request from the NETBIOS. As a result, we can not make any assumptions about the state of the system (other than the value of **ES:BX**). The code given in **(post)** uses the address stored in the **TASK@** field of the NCB to discover which task is related to the NCB. It will then place a 1 in that task's **STATUS** variable, thereby adding that task to the scheduler active task list.

```
CREATE-INTERRUPT (post)
  DS PUSHSEG   BX PUSH   AX PUSH   ES AX MOV   AX DS MOV
  FIELD TASK@ 2+ ) BX@ AX MOV   AX PUSH
  FIELD TASK@ ) BX@ AX MOV   AX BX MOV
  DS POPSEG   1 # USER STATUS MOV
  AX POP   BX POP   DS POPSEG
IRET
END-CODE
```

This code is given as it is provided in the FORTH++ interface. We now give the code again in a commented Int$_e$l assembler format.

```
post: push ds          ; Save the registers
      push bx          ;   we are going to use
      push ax

      mov  ax,es       ; Copy ES to DS
      mov  ds,ax

      mov  ax,[bx+66]  ; Get the DS for the task
      push ax          ; Save it for later

      mov  ax,[bx+64]  ; Get the offset of the task

      mov  bx,ax       ; Save in BX
      pop  ds          ; Recover task's DS

      mov  [bx+0],#1   ; Set task's status to active

      pop  ax          ; Recover registers
      pop  bx
      pop  ds

      iret             ; Return from interrupt
```

The next word we define is **(net)**. This word will initialise the NCB with a given command (**CMD**), buffer (**BUFFER@**) and post routine (**POST@**). It will then invoke the NETBIOS interrupt asking the NETBIOS to perform the function indicated by the command number. The **POST@** value passed to this word is the 16 bit offset of the **(post)** routine. If this offset is 0, an address of **0000:0000** is placed in the **POST@** field. When the NETBIOS returns from the interrupt it provides a "return value" that is passed back to the calling word.

```
HEX

CODE (net) ( NCB Buffer Command 'Post -- Retcode )
  CX POP   AX POP   DX POP   DI POP
  AL FIELD CMD ) DI@ MOV   DS AX MOV
```

```
  AX FIELD BUFFER@ 2+ ) DI@ MOV   DX FIELD BUFFER@ ) DI@ MOV
  CX AX MOV   0 # AX = NOT        IF  CS AX MOV  THEN
  AX FIELD POST@ 2+ ) DI@ MOV     CX FIELD POST@ ) DI@ MOV
  DS AX MOV
  AX FIELD TASK@ 2+ ) DI@ MOV     BX FIELD TASK@ ) DI@ MOV
  ES PUSHSEG   BX PUSH   DS AX MOV   AX ES MOV   DI BX MOV
  5C INT   BX POP   ES POPSEG   0 # AH MOV   AX PUSH
NEXT
END-CODE

DECIMAL
```

Again, this code is given as it is provided in the FORTH++ interface. We now give a version of the same code, with comments, in Int$_e$l assembler format.

```
net:  pop   cx             ; CX = POST@ offset
      pop   ax             ; AX = NetBios command
      pop   dx             ; DX = BUFFER@ offset
      pop   di             ; DI = NCB offset

      mov   [di+00],al     ; Set NetBios command in the NCB

      mov   ax,ds
      mov   [di+06],ax     ; Set the BUFFER@ segment to the current DS
      mov   [di+04],dx     ; Set BUFFER@ to the given offset

      mov   ax,cx          ; Is POST@ offset zero?
      cmp   ax,#0
      jne   $1             ; Yes, then AX and CX = 0
      mov   ax,cs          ;  No, then set AX to current CS

$1:   mov   [di+46],ax     ; Set POST@ segment to CS (0000 if CX=0000)
      mov   [di+44],cx     ; Set POST@ offset to CX

      mov   ax,ds
      mov   [di+66],ax     ; Set TASK@ segment to current DS
      mov   [di+64],bx     ; Set TASK@ offset to task user area

      push  es             ; Save registers ES:BX
      push  bx
      mov   ax,ds
      mov   es,ax          ; ES:BX = NCB address
      mov   bx,di
      int   5Ch            ; Invoke NetBios interrupt

      pop   bx             ; Recover ES:BX
      pop   es

      mov   ah,#0          ; Clear top byte of "Return Value"
      push  ax             ; Return "Return value"

      NEXT                 ; Re-enter inner interpreter
```

2.8.4 Low-Level interface

The next part of the interface defines the low-level FORTH words that are used to interface with the assembler definitions.

The first of these words is **+NET**. It will post a NETBIOS function and wait for it to complete before returning. It will then process the "Return Value", checking it for errors.

```
: +NET ( Buffer NCB Command -- )
```

```
  ROT SWAP 0 (net) NETERROR
;
```

The second word being **-NET** which will post a network function to the NETBIOS system using the *No-Wait with Interrupt* variant of the command. The calling task will be placed in the scheduler's active list on completion of the function. However, the task is not removed from the active list by this word. This is left to the application.

```
: -NET ( Buffer NCB Command -- )
  128 OR ROT SWAP (post) (net) NETERROR
;
```

We now define the word **COMPLETE** to check the NCB command complete (**CMD_CPLT**) flag. It will return a **TRUE** when the function has completed. This word is provided so that an application may test which of several possible NETBIOS commands has been honoured (see sections 2.4 and 2.5.2 for a description of its use and section 2.9.2 for an example of its use).

```
: COMPLETE ( NCB -- f )
  CMD_CPLT C@ 255 = NOT
;
```

The final definition in this section is **NERROR** which is used in conjunction with the **COMPLETE** word. It will check the return code (**RETCODE**) of a given NCB returning the NETBIOS return code, if the function associated with the NCB has completed, otherwise it returns a **-1**.

```
: NERROR ( NCB -- n )
  DUP COMPLETE IF RETCODE C@ ELSE DROP -1 THEN
;
```

2.8.5 General Support

Here we define a number of words for the general administration of the network. Most of these commands would only be used by a supervisor or supervising software. These commands do not have *No-Wait* variants, thus they all wait for the NETBIOS command to complete before returning to the caller.

NET-RESET will *Reset* the network with the support for the given number of sessions and the given number of outstanding commands using the given NCB.

```
: NET-RESET ( #sessions #commands NCB -- )
  DUP >R NUM C! R@ LSN C! 0 R> 50 +NET
;
```

NET-CANCEL is used to *Cancel* a NETBIOS command. The NETBIOS command associated with **NCB1** is cancelled (removed from the command-pending list). Due to the way that the NETBIOS system operates, it requires a second NCB to be used to issue the cancel command.

```
: NET-CANCEL ( NCB1 NCB2 -- )  53 +NET ;
```

The **UNLINK** word will disconnect the node from the "Remote Program Link". This is only used when booting the system over a network.

```
: UNLINK ( NCB -- )       DUP 112 +NET ;
```

Finally the **NET-STAT** word returns the current status of the network to the given buffer (**addr**) of a given maximum size (**len1** bytes). Returning the number of bytes (**len2**) of actual data received. This data is dependent on both the network hardware and the particular NETBIOS implementation.

```
: NET-STAT ( addr len1 NCB -- len2 )
  SWAP OVER LENGTH DUP >R ! DUP CALLNAME ASCII * SWAP C!
  51 +NET R> @
;
```

2.8.6 Naming Support

In this section we define the FORTH words that will give the programmer access to the NETBIOS "Naming" functions.

Firstly, the word **(name)** is defined. This word takes a counted string (**s**) as a symbolic name. It will place the name in the given **NCB**'s **NAME** field. This takes a fixed 16 character name, thus **(name)** also pads out the field with zeros. Having copied the name into the **NAME** field, it will then invoke the NETBIOS function given in **n** (either *Add Name* or *Add Group Name*). Notice that it uses **+NET** to invoke the function, thus the system will wait for the name to be added to the local name table before returning. This word forms the bases of both the **ADD-NAME** and **ADD-GROUP** words.

```
: (name) ( s NCB n -- )
  >R DUP NAME DUP 16 ERASE ROT COUNT ROT SWAP CMOVE
  0 SWAP R> +NET
;
```

The word **ADD-NAME** is used to add an individual name to the list of logical names for this node. It takes a counted string (**s**) and an **NCB**. It will add the name to the system, associating the name with the NCB. Any command sent out using that NCB will be issued under the given name. You must copy the NCB if you wish to post more than one (simultaneous) command under this name.

```
: ADD-NAME  ( s NCB -- )  48 (name) ;
```

The **ADD-GROUP** command works in much the same way as the **ADD-NAME** command with the one exception that the name added to the local node is a group name. Thus several different nodes may be known by the same name.

```
: ADD-GROUP ( s NCB -- )  54 (name) ;
```

The final word in this section is **REMOVE-NAME**. This will remove the name associated with the **NCB** from the local name table. If the NCB is associated with a group name, the node is removed from the group. The name is disassociated from the **NCB**, thus allowing the **NCB** to be associated with another name.

```
: REMOVE-NAME (   NCB -- )  0 SWAP 49 +NET ;
```

2.8.7 Session Support

In this section, we provide words that allow the application programmer to access the session handling facility of the NETBIOS.

Before we define the words that the application programmer is to use, we first define two words that perform most of the operations. These words are internal to the interface and are not meant to be used by the application programmer.

The first of these is **(cname)** which takes a counted string (**s**) and places it in the **CALLNAME** field of the given **NCB**. As with the **(name)** word, this also pads the field out to 16 characters by adding zeros. **(cname)** not only leaves the NCB address on the stack, it also places a 0 onto the stack to be used as a null buffer address. See the words **PHONE** and **LISTEN** to see how the word is used.

```
: (cname) ( s NCB -- 0 NCB )
  DUP CALLNAME DUP 16 ERASE ROT COUNT ROT SWAP CMOVE 0 SWAP
;
```

The second internal word is **(len)**. This will simply place the given buffer length (**len**) into the **LENGTH** field of the given **NCB** without removing the NCB address from the stack.

```
: (len) ( len NCB -- NCB )
  SWAP OVER LENGTH !
;
```

Having defined the two supporting words, we can now go on to define the words that the application programmer will use to gain access to the NETBIOS session capability. As we have already likened a session connection to a telephone connection, we use telephone-like words in our interface.

The word **PHONE** is used to establish a connection. This is similar to making a telephone *call* where you give the name of the recipient as a counted string (**s**). If the call is being made to a group name, only one member of the group will receive the call. The NETBIOS selects the group member, a one-to-one connection is made with one of the group members. The particular member is not known and is non-deterministic.

```
: PHONE     (        s NCB --      )   (cname) 16 -NET ;
```

The word **LISTEN** is similar to listening for a telephone call. You give the name of the node you are waiting to hear from as a counted string (**s**). However, you will only hear calls from that node, if another node is attempting to contact this name, the *listen* command will not register the call. When a call is detected, a connection (session) is established on both nodes.

There is a special name of "*" that will listen for a call from anyone. When a call is detected, the session (connection) is established and the name of the caller is placed in the **CALLNAME** field of the NCB.

```
: LISTEN    (        s NCB --      )   (cname) 17 -NET ;
```

The word **HANGUP** is used to disconnect the session. This is similar to someone hanging up the telephone to break the connection. We use the same convention as is used for telephones in that the caller is responsible for clearing the connection.

```
: HANGUP    (          NCB --      )    0 SWAP 18 -NET ;
```

We now have the words that will allow one to set up a connection but we are still unable to transfer data over this connection. The next two words provide this capability. The connection must be established prior to any attempt to transmit data.

To transmit data over the connection (to source the data) we use the **TX** word. This takes a buffer (**buff**) of **len** bytes (the maximum buffer size being 64 KBytes) and transmits it over the connection. As this is a session connection NETBIOS provides a guarantee that the data will arrive.

```
: TX        ( Buff Len NCB --      )     (len) 20 -NET ;
```

To sink (receive) the data the **RX** word is used. We give the system a buffer area (**buff**) with a maximum size of **len** bytes where it can place the data when it is received. When data has been received, the **LENGTH** field of the NCB will hold the actual number of bytes received. If the buffer is not large enough to hold all the data, the system will buffer the remaining data internally and report an error. Under these conditions an error code of 6 is placed in the **RETCODE** field of the NCB. It is the responsibility of the application programmer to detect and act on this condition by issuing another receive request.

```
: RX        ( Buff Len NCB --     )     (len) 21 -NET ;
```

The final word in this section is **CALL-STAT** which is used to obtain status information on the connection (session) associated with the given **NCB**. It is given a buffer (**buff**) of **len1** bytes into which it will place the current status. The **CALL-STAT** word will return the actual number of bytes used (**len2**) by the status information. The status information returned by this word is partly defined, however a large part of the data is dependent on the NETBIOS implementation.

```
: CALL-STAT ( Buff Len1 NCB -- Len2 )
  SWAP OVER LENGTH DUP >R ! 52 +NET R> @
;
```

2.8.8 Datagram Support

This is where we develop the FORTH words that will give the application programmer access to the "Datagram" communication level provided by the NETBIOS. A datagram can be thought of as a packet of up to 512 bytes on the network. Unlike session communication, there is no built-in protocol associated with datagrams. The receiving node *must* be listening for an incoming datagram, otherwise it will not receive it. The NETBIOS provides no guarantee that the datagram will be delivered.

The first word we define in this section is **DTX**, the *Datagram Transmit* function. This will take an area of memory (**buff**) of **len** bytes in length (maximum size being 512 Bytes). This is sent, as a single unit, to the indicated node (whose name is given as the counted string **s**).

Notice how this word uses **(cname)** to copy the destination node name into the **CALLNAME** field of the NCB. The **NIP** is required to disregard the extra **0** that **(cname)** places on the stack. We use **(len)** to copy the byte length into the **LENGTH** field of the NCB. We can make the NETBIOS call with the **-NET** word.

```
: DTX ( Buff Len s NCB -- )
  (cname) NIP (len) 32 -NET
;
```

The *Datagram Receive* function is provided by the word **DRX**. This is given an area of memory to place the received data (**buff**) which is a maximum size of **len** bytes (maximum buffer size is 512 bytes). This word will wait for an incoming datagram addressed to the name associated with the NCB. On receiving a datagram, it will place as much data as it can in the buffer returning the actual number of bytes received in the **LENGTH** field of the NCB. Note that if the received datagram was too large for the receiving buffer, the buffer is filled, the remaining data is lost, and a return value of 6 is given (in the **RETCODE** field). The name of the sending node is placed in the **CALLNAME** field. See section 2.9.2 for an example of using datagrams.

```
: DRX ( Buff Len NCB -- )  (len) 33 -NET ;
```

2.8.9 Broadcast Support

In this, the final part of the interface, we define the words that provide access to the NETBIOS "Broadcast" commands. A broadcast can be thought of as sending a datagram to everybody. If you are

not listening for a broadcast, you will miss it. Like the datagram it will not be buffered for you. As with datagram support, we only need two words to provide broadcast support, one to transmit and one to receive.

The first of these words is **BTX**, providing the *Broadcast Transmit* function. This takes the address of the memory buffer (**buff**) of **len** bytes in length (maximum size of 512 bytes). The data is then transmitted to every node on the system.

```
: BTX ( Buff Len NCB -- )  (len) 34 -NET ;
```

The second word required to provide broadcast support is **BRX**, providing the *Broadcast Receive* function. As with **DRX**, the address of a receive buffer is given (**buff**) with a maximum length of **len** (maximum buffer size is 512 bytes). When the system receives a broadcast message, it will place up to **len** bytes in the buffer loosing any additional data. The **LENGTH** field holds the actual number of bytes received. The **CALLNAME** field will hold the name of the sending node. If more than one *Broadcast Receive* is posted, they will *all* receive the same message.

```
: BRX ( Buff Len NCB -- )  (len) 35 -NET ;
```

It should be noted that the words **(netable)**, **(neterror)**, **pos**, **FIELD**, **(post)**, **(net)**, **+NET**, **-NET**, **(name)**, **(cname)** and **(len)** are internal to the interface and should not be used when programming applications with this package.

2.9 The "Net-Chat" Application

The following is an annotated source listing of the "Net-Chat" example application as described in Section 2.5.2.

2.9.1 Memory Buffers

The first part of the application is to reserve the memory buffers that are going to be used. This section not only reserves the memory but also defines words that allow easy access to this memory.

We are going to require five NCBs and buffers. We first reserve the space for the five NCBs (one outgoing, four incoming). The number of bytes to reserve is calculated by multiplying the number of bytes required for an NCB (**ncb_size**) by five. We then initialise this memory to zeros using the **ERASE** word.

```
CREATE ncbs   ncb_size 5 * ALLOT   ncbs ncb_size 5 * ERASE
```

Thus the word **ncbs** will return the start address of a block of memory large enough to hold five NCBs. We now define a word **NCB** that take an NCB number and returns the address of the indicated NCB from our table.

```
: NCB  ( n -- NCB )       ncb_size * ncbs + ;
```

Now we do the same for the data buffers. This time the buffers are 60 bytes long and is given the name **buff**, while the accessing word is called **BUFF**.

```
CREATE buff         60 5 * ALLOT    buff  60 5 *  ERASE

: BUFF ( n -- buff )      60 * buff +       ;
```

We now define the word **name** that takes an NCB number and initialises the stack ready for a NETBIOS call to the *Datagram Receive* function, placing the corresponding buffer address (**buff**), the maximum size of the buffer (60) and the indicated NCB (**NCB**) on the stack.

```
: name ( n -- buff 60 NCB )
  DUP BUFF SWAP NCB 60 SWAP
;
```

2.9.2 Listening

In this section we define the "Listening" part of the application. This code will post four *Datagram Receives* to the NETBIOS and wait for one of them to be honoured. It will then display the name of the sender and a one line message.

The first item to define is the actor that is going to execute the code (**CHAT-TASK**). The actor is defined now so as to indicate that all the code that follows (upto the **CONSTRUCT** word) will be performed by the actor concurrently with the main system.

```
ACTOR CHAT-TASK
```

The first word we define in the section is **NET-LISTEN** which simply posts four *Datagram Receive* functions which will operate in unison. It should be noted that NCB 0 has been reserved for outgoing messages.

```
: NET-LISTEN
  5 1 DO
      I name DRX
  LOOP
;
```

When one of these *Datagram Receive* functions has been honoured, the system will execute the **NET-DISP** word. This will scan through the NCBs to discover which of them has been honoured. It will then display the name of the sender (taking it from the **CALLNAME** field) and the associated message. Finally it re-posts the *Datagram Receive* command.

```
: NET-DISP
  5 1 DO                              \ Scan through the incoming NCBs
    I NCB COMPLETE                    \ Has the command been honoured ?
      IF
         I NCB CR
         CALLNAME 16 0 DO             \ Display the CALLNAME filed
           DUP C@ ?DUP 0= IF LEAVE THEN EMIT 1+
         LOOP DROP
         ." : "                       \ Display a name separator
         I BUFF I NCB LENGTH @ TYPE   \ Display the message
         I name DRX                   \ Re-post the DRX
      THEN
  LOOP
;
```

The output from **NET-DISP** will be displayed in a small window at the top of the screen. The following line defines the window to start at the top left of the screen, being 78 characters wide and 5 lines high. The **WITH-BORDER** indicates that the window will have a line boarder displayed around it. Finally the window will be called **NET-WIN**.

```
1 1 78 5 WITH-BORDER CREATE-WINDOW NET-WIN
```

The last word to be defined in this section is **NET-GO**. This is the word that the **CHAT-TASK** will be asked to perform (by the **GO** word). It initialises the window and posts the initial four *Datagram Receive* requests. It then enters into an infinite loop waiting for one (or more) of the requests to be honoured when it will call the **NET-DISP** word to display the message and re-post the receive request.

```
: NET-GO
  NET-WIN <WIN            \ Open the window.
    *WCLEAR               \ Clear it
    *TITLE"  Net Chat "   \ Give it a title

    NET-LISTEN            \ Post initial four DRX commands
    BEGIN
      STOP                \ Wait for one to be honoured
      NET-DISP            \ Display the message & re-post
    AGAIN
  WIN>
;
```

The final act in this section is to indicate the completion of the code that is to be executed by the **CHAT-TASK** actor. This also completes the definition of the actor. Any words defined from this point on would not be accessible to the **CHAT-TASK** actor.

```
CHAT-TASK CONSTRUCT
```

2.9.3 Sending

In this section we define the "Sending" part of the application. In reality this consists of one definition. The word **CHAT** will ask the user to type in a one line message. It will then send the message as a datagram to the group name "**NET-CHAT**", thus any node with a *Datagram Receive* posted on the group name **NET-CHAT** will receive a copy of the message (including the sending node).

Firstly, the word locates the outgoing message buffer (buffer 0). It then erases the buffer making sure no other message is stored there. It now displays a message asking the user to input the message they wish to transmit. The message is read directly into the buffer with a maximum of 60 characters in length:

	78	Characters in the display line
–	16	Maximum characters in user name
–	2	Name/Message separator (": ")
	60	Total allowable size of message

The number of characters actually typed is taken as the size of the buffer. The buffer is sent to the group name **NET-CHAT** via the outgoing NCB (NCB 0). Finally, the word waits for the *Datagram Transmit* function to complete before returning to the user.

```
: CHAT
  0 BUFF                              \ Find outgoing buffer
  DUP 60 ERASE                        \ Erase buffer
  CR ." Message: "                    \ Ask for the message
  DUP 60 EXPECT                       \ Read in the message
  SPAN @  " NET-CHAT"  0 NCB  DTX     \ Send the message
  STOP                                \ Wait for NetBios to complete
;
```

2.9.4 Initialisation

In this part of the application, we provide the initilisation of the system. The word **GO** initialises the system for use with the "Net-Chat" application as outlined in section 2.5.2.

The first part of the initialisation is to define a word that is going to become the network error handler for the application. This is a very simple word that simply ignores any errors. This definition is required so that the **INIT-CHAT** word can examine the return code and take appropriate action. (The default action will cause the system to abort on an error.)

```
: NO-ERROR  DROP ;
```

The next part of the initialisation process is coded into the word **INIT-CHAT**. This initialises the network handling side of the system. Firstly, it replaces the standard error handling with our error handling system (**NO-ERROR**). It will then ask the user to type in a unique name that it will use to identify the user to the other uses of the system. It attempts to add the name to the network (*Add Name*). If an error occurs a message is displayed and the user is asked to supply an alternative name.

When the individual name has been established (on the outgoing NCB, NCB 0), the error handler is reset back to the default. The **NO-ERROR** handler is only used to allow the word to extract the error code and ask for another name if necessary.

The group name **NET-CHAT** is added to the network (on NCB 1). The information placed in the NCB by the *Add Group Name* function is copied to the remaining incoming NCBs (2, 3 and 4).

```
: INIT-CHAT
  'NETERROR @                       \ Save the default error handler
  ['] NO-ERROR  'NETERROR !         \ Reset the error handler

  BEGIN
    CR ." Enter your name: "        \ Ask for a name
    0 BUFF DUP 1+ 16 EXPECT         \ Read the name (max 16 chars)
    SPAN @ SWAP C!                  \ Make buff a counted string
    0 BUFF 0 NCB ADD-NAME           \ Add name to Network
    0 NCB NERROR
  WHILE                             \ While error in Add-Name
    CR ." Sorry, someone else is already using that name, try another."
  REPEAT                            \ Repeat input sequence

  'NETERROR !                       \ Reset error handler to default

  " NET-CHAT"  1 NCB  ADD-GROUP     \ Add the group name
  1 NCB DUP DUP
  2 NCB COPYNCB                     \ Copy the NCB data to NCB 2
  3 NCB COPYNCB                     \ '' NCB 3
  4 NCB COPYNCB                     \ '' NCB 4
;
```

The window for use by the **OPERATOR** actor is now defined to be 15 lines of 78 characters starting at line 8, complete with a line boarder.

```
1 8 78 15 WITH-BORDER CREATE-WINDOW OP-WIN
```

Finally, the word **GO** is defined. This is the word that the user will type to initialise the "Net-Chat" application.

The first action of **GO** is to call the **INIT-CHAT** word. Thus it asks for an individual name and initialise the NCBs. **GO** will then clear the screen (**CLEAR**) and turn the hardware cursor off (**HWC-OFF**) ready for the windowing environment. It will then redirect the **OPERATOR** output to the **OP-WIN** window (**<WIN**). Finally, the actor **CHAT-TASK** is sent the message (**SEND"**) to initialise its window and listen for and display incoming messages (**NET-GO**).

```
: GO
  INIT-CHAT                 \ Initialise the Network
  CLEAR                     \ Clear the screen
  HWC-OFF                   \ Turn the hardware cursor off
```

```
  OP-WIN <WIN                \ Redirect output to the OP-WIN window
  *WCLEAR                    \ Clear the window
  *TITLE"  Operator "        \ Title the window
  CHAT-TASK SEND" NET-GO "   \ Set the CHAT-TASK listening
;
```

2.9.5 Close Down

In this, the final section of the application, we provide the code that will close down the application. All applications should provide a graceful close down, especially when they are using the services of some kind of server such as the NETBIOS.

There are a number of things we need to do to close down: stop the `CHAT-TASK` actor; remove the unique name from the system; cancel any outstanding commands; resign from the `NET-CHAT` group; tidy up the screen. The order in which these events occur is quite important. All of this can be accomplished in the one FORTH word, `CLOSE-CHAT`. This is the word that the user will type when they wish to close or leave "Net-Chat".

Our first task is to force the `CHAT-TASK` actor to stop processing. This we do by forcing it to accept a new task (via the `MUST SEND"` operation). We ask `CHAT-TASK` to close its window (`WIN>`) and then to stop processing until further notice (`HALT`). Having stopped `CHAT-TASK` from receiving any messages, we are now able to alter the status of the network. We first remove the unique name from the name table (`REMOVE-NAME`). This provides us with a free NCB which we use to cancel the *Datagram Receive* requests that `CHAT-TASK` would have posted (`NET-CANCEL`). Notice how any task can cancel these requests as the NETBIOS is unaware of our tasking mechanism, thus does not consider a NETBIOS request to be owned by any particular task.

We are no longer able to send a message as we do not have a unique name. We are no longer able to see messages as `CHAT-TASK` is not running. We are no longer listening for messages sent to the `NET-CHAT` group as we have just cancelled all such requests. Thus we are now in a position to be able to resign our membership of the `NET-CHAT` group (`REMOVE-NAME`). Finally, we close the operations window (`WIN>`) and re-establish the cursor (`HWC-ON`).

```
: CLOSE-CHAT
  CHAT-TASK MUST SEND" WIN> HALT"     \ Close NET-WIN and stop the task

  0 NCB REMOVE-NAME                   \ Remove the outgoing unique name

  1 NCB  0 NCB  NET-CANCEL            \ Cancel the DRX commands
  2 NCB  0 NCB  NET-CANCEL
  3 NCB  0 NCB  NET-CANCEL
  4 NCB  0 NCB  NET-CANCEL

  1 NCB REMOVE-NAME                   \ Remove the group name

  WIN>                                \ Close OP-WIN
  HWC-ON                              \ Turn hardware cursor on
;
```

If the user wanted to restart the application, he would simply type `GO` and he would be back in the application.

Chapter 3

Mixed Languages interface

In this chapter we describe a mixed languages interface developed for use between the C and FORTH languages. The general ideas and principles used in developing this code are independent of the FORTH and C systems being used. This interface has been compiled under a number of different systems including Borland's "Turbo C", the ZorTech C and C++ compilers in addition to the Microsoft C compiler.

3.1 Principles

The basic principle of the interface is that sufficient state information is stored when switching between FORTH and C operations for both languages to appear to be in full control of the system. The system starts with the C main program which loads and executes the FORTH system. Control will now stay with the FORTH system until such time that FORTH passes control back to C. At this point, C sees the FORTH parameter stack as a simple data structure. The C code will *pop* an item off the FORTH stack and use it as an index into a function table. The requested function is then executed and control is returned to FORTH.

3.2 Argument Passing

All of the argument passing between the two languages is performed by the C system manipulating the FORTH parameter stack as a data structure. Several C functions have been defined to manipulate the FORTH stack. These include operations to *drop* an item, *pop* an item, *push* a value and a function that allows us to *index* into the stack.

In order to make this system more usable, these functions have been defined as type independent macros, thus they take a type indicator as an argument. To pop an integer off the stack we would write the statement "`x = POP(int);`" and to push a floating point value onto the stack the statement "`PUSH(float, n);`" would be used.

3.3 Programming

In order to show the way in which you would program some code, let us look at an interface to the memory allocation system (C heap management).

The code fragment in figure 3.1 is placed in the users C file. A reference to this function must be placed into a jump table (fig 3.2).

In the FORTH system, a word has been defined that calls the C code via a vector address. To execute this function you would define a FORTH word such as `GETMEM` as shown in figure 3.3.

```
getmem()
{
  void *ptr;
  int size;

  size = POP(int);
  ptr = (void *)malloc(size);
  PUSH(void *,ptr);
}
```

Figure 3.1: The C `getmem` function.

```
TBL jmptbl [] =
{
  ...
  /* Function 8 */  getmem,
  ...
}
```

Figure 3.2: Example jump table.

This would be used as '`1000 GETMEM`'. The C code *pops* the value (1000) into an integer variable (`size`). It will then return a pointer to the memory (`ptr`) allocated by the C system call.

In our implementation, we have defined two FORTH words `CCALL` and `-CCALL` to handle C function calls. Suppose that we wanted to define words to access the function table as given in figure 3.4, we would write the code given in figure 3.5.

The FORTH words `ARC` and `CIRCLE` are defined to call the C code with the relevant function number. However, the `BAR` function is used as a place holder. If at some time in the future we wish to define the `BAR` word, we would simply remove the `-` from the `-CCALL` that is holding `BAR` in place.

3.4 The C Heap

The C system assumes that it has full control of the system memory. Due to this assumption, we must take care when deciding how to load the FORTH system into memory.

The correct method is to request space for the FORTH system from the C heap. FORTH should request memory only via a call to the C system. If the FORTH system were to invoke the memory management system calls directly, this may cause the C heap to become invalid.

This is particularly relevant with regard to MS-DOS where some of the C systems we have used attempt to expand their heap by resizeing the memory space allocated to it rather than by requesting a fresh memory area. If an area of memory allocated to FORTH prevents this operation, the C system will

```
: GETMEM 8 CCALL ;
```

Figure 3.3: The `GETMEM` word.

```
TBL jmptbl [] =
{
   /* Function 1 */  draw_arc,
   /* Function 2 */  draw_bar,
   /* Function 3 */  draw_circle
}
```

Figure 3.4: Another example jump table

```
 CCALL ARC
-CCALL BAR
 CCALL CIRCLE
```

Figure 3.5: Example of using `CCALL` and `-CCALL` to define words

incorrectly assume that it has run out of available memory.

3.5 Organisation

Our system has been split into three modules. The first of these is the main C module that holds the C `main()` function. A second module was written to hold non-portable code (this loads the FORTH system and handles the transfer of control between the two systems). Neither of these modules should be changed by the user. We have placed them into a C library file to be linked in with the third module supplied by the user.

This third module holds all of the users C code and the function jump table. The user compiles this module and links it with the required libraries (including ours) to produce a new C base program.

The source of this interface is far too large to be included here. The full, documented, sources to the interface (including various development macros or scripts), along with a number of technical comments about the system is given, in Appendix C.

3.6 Generalisation

We have made our implementation as general as possible. However, the two routines to load and initialise the FORTH system and to transfer control between the two systems have to be specific to the systems being used.

In our implementation, it is assumed that all memory references are in long (or `far`) form. Thus, it is necessary that all the modules be compiled using the *large* memory model. This is a restriction imposed on use by using a segmented memory structure.

All of the files have been written using standard coding. The FORTH code is a very simple change to the compiled system. The C code has been written to the ANSI standard while the assembler code is written using the standard Microsoft assembler.

Chapter 4

Formal FORTH

In this chapter, we present a system that will aid in ascertaining whether a program meets its specification. By providing a formal base for the FORTH language, we can formally discover the coherence between the specification and its implementation, thus we have the ability to prove that the program meets its specification.

4.1 Introduction

The conventional computer science approach to programming languages starts by separating syntax from semantics.

The syntax deals with allowable statements or sentence formation and has been investigated using techniques that apply equally well to simplified forms of natural language. These techniques result in a classification of languages into categories such as phrase structured, context sensitive and context free. A powerful body of theory (and application) has built up around the syntax of a language.

The semantics of a language deals with the meaning of program text; the interpretation that is placed on a syntactically correct phrase in a given language.

Most language definitions have a formal description of the grammar that describes syntactically correct statements for the given language, however, the syntax of a FORTH system is semantically defined. You could say that FORTH is not a computer language, rather a *dictionary of words* where each word has a *definition* which describes the operation it performs in terms of existing definitions or in terms of the native code of the machine on which the system is implemented.

The set of words thus defined perform all operations executed by the system including the scanning of FORTH text to be compiled or interpreted. A word may be defined to ignore or amend following words in the input stream. It is these abilities that make it difficult to apply classical syntax theory to FORTH.

Many compiler developers use a virtual machine similar to the FORTH abstract machine as a universal intermediate code (Cook and Lee 1980; Miller 1987). Thus, one could say that the FORTH abstract machine is the ideal computer model (Kavipurapu and Cragon 1980).

4.2 The FORTH Toolbox

In order to talk formally about a program, we must have a formally described programming language/environment. FORTH provides us with a simple language with a programming environment and debugger. Due to the simple nature of FORTH, this can be formalised much more readily than most other languages (Stoddart 1988). The formal description of the FORTH programming environment will provide us with an additional toolbox to use when formally describing an application program.

4.3 The Basic Model

Let us assume that we have a set of all of the known memory locations in a system and that we have a set of all the possible (allowable) names for a FORTH system:

$$[ADDRS,\ NAMES]$$

It is possible to say that the FORTH dictionary is a relation between names and addresses. However, defining a simple relation does not capture the ordered (historical) nature of the dictionary so we make this a sequential relationship:

$$dict : \mathrm{seq}\,(NAMES \times ADDRS)$$

An example entry of this type would be $(6, (\text{“}\texttt{@}\text{”}, 204))$ where the FORTH word “`@`” is the 6^{th} entry in the dictionary and has an address of 204, quite how this is implemented is unimportant. In a token based system, the 6 could be thought of as being the token for the word while a threaded code system may not store the 6 at all but uses the associated address. For our purposes we will use the notion of a token:

$$token : \mathbb{N}$$

4.4 Word Definitions

We now have the sequence *dict* that tells us what words are in the dictionary and where they can be found. We have yet to record the definition of a given word, to do this we introduce a function that relates known words to their definitions:

$$body : token \rightarrow \mathrm{seq}\ NAME$$

where *token* is the index number of the word in the dictionary and seq *NAME* is the sequence of words that make up the definition. Hence, a word such as `NIP` may have a dictionary entry of[1]:

$$\{33, (\text{“}\texttt{NIP}\text{”},\ 378)\}$$

and a definition of:

$$\{33 \mapsto \langle \texttt{SWAP},\ \texttt{DROP} \rangle\}$$

4.5 Immediate Words

We must be able to discover if a word is immediate or not. Hence, we introduce a function taking the *token* of a word and returning a true if the word is immediate or a false if not:

$$immediate : token \rightarrow \{\mathrm{true}, \mathrm{false}\}$$

4.6 Storage Units

FORTH does not use types in the conventional manner. Instead of types it uses classes of storage unit. There are three classes of storage unit: *Character*, *Cell* and *Double Cell*.

Each class of storage unit is able to store any number of types that the application program requires. The only limitations being hardware restrictions. The application programmer may add to the list of possible types that a given storage class can hold, indeed he can even add new storage classes.

[1] Note that the addresses and token values given in this chapter are for example only and do not relate to any given system.

Words are defined with reference to the unit class rather than the exact type required. If we were to enforce the use of types in our model we would not be modelling the full behaviour of a FORTH system. Hence this system uses the notion of classes of storage unit.

We must introduce the classes of storage unit as given sets of types:

$$[Char,\ Cell,\ DoubleCell]$$

4.7 Stacks

We must provide a mechanism for the parameter and return stacks. This we do by defining two global variables consisting of a sequence of stack cells:

$$pstack, rstack : \text{seq}\ Cell$$

Thus we could define the FORTH word `DEPTH` as:

$$\texttt{DEPTH} \mathrel{\widehat=} [\, pstack' = \langle \# pstack \rangle \frown pstack \,]$$

Ie, we push onto the stack (add to the start of the sequence) the size of the stack (sequence) as it was at the start of the statement. It should be noted that $pstack'$ is the standard way of indicating the state of the parameter stack after the operation while $pstack$ refers to the state of the parameter stack prior to the operation.

A possible definition for `DROP` would be:

$$\texttt{DROP} \mathrel{\widehat=} [\, pstack' = tail\ pstack \,]$$

Ie, the stack (sequence) now holds all that was previously on the stack (in the sequence) except for the top most (first) element.

Our system will also have to cater for words with variable stack effects such as the `?DUP` word. We can represent this by placing a side condition on the stack description. Ie, `?DUP` is defined as:

$$\texttt{?DUP} \mathrel{\widehat=} \left[\begin{array}{c} \Big((pstack' = pstack) \land (pstack(1) = 0) \Big) \\ \lor \\ \Big((pstack' = \langle pstack(1) \rangle \frown pstack) \land (pstack(1) \neq 0) \Big) \end{array} \right]$$

So far we have only discussed words that effect the parameter stack ($pstack$). However, the system is sufficiently flexible, we can define words such as `>R` which not only effect the parameter stack but also the return stack ($rstack$). The definition of `>R` would be:

$$\texttt{>R} \mathrel{\widehat=} \left[\begin{array}{c} pstack' = tail\ pstack \land \\ rstack' = \langle pstack(1) \rangle \frown rstack \end{array} \right]$$

while the definition for `R>` would be:

$$\texttt{R>} \mathrel{\widehat=} \left[\begin{array}{c} pstack' = \langle rstack(1) \rangle \frown pstack \land \\ rstack' = tail\ rstack \end{array} \right]$$

4.8 Code Definitions

There are many words that are coded in the native machine language of the host computer, `SWAP` and `DROP` are two such words. In order to cater for such words, we introduce a set of code level words. As these words are defined in the native machine language, we can not give their definitions, however, we can give a formal description of the function that they perform.

Assuming that the words `SWAP` and `DROP` have the following dictionary entries:

$$(3, (\text{“}\texttt{SWAP}\text{”}, 30)) \text{ and } (4, (\text{“}\texttt{DROP}\text{”}, 36))$$

then we could represent there actions as:

$$\left\{3 \mapsto [\, pstack' = \langle pstack(2), pstack(1)\rangle \frown tail\, tail\, pstack \,]\right\}$$

and

$$\left\{4 \mapsto [\, pstack' = tail\, pstack \,]\right\}$$

So we now have a function that relates known “code-level” words to their required action:

$$code : token \rightarrow axiom$$

Thus giving us an additional set of *axiom*s to work with when reasoning about the implementation. Thus the dictionary is split into “high-level” (*body*) or “code” (*code*) words.

$$\mathrm{dom}\, body \cap \mathrm{dom}\, code = \varnothing$$

It should be noted that we have not provided an instruction pointer. To do so would be to restrict the number of possible implementation techniques. Although the FORTH abstract machine calls for an instruction pointer, we leave it up to the implementor to introduce and define their own.

A consequence of this is that operations such as `NEXT`, `EXIT` and the run-time action of `:` and `;` are currently not specifiable. They must be provided by the implementor, thus allowing them to model their particular method of implementation.

4.9 Wordlists

We define a set of wordlists so that the dictionary is composed of several wordlists, where the wordlists include all the entries in the dictionary. Yet no single entry occurs in more than one wordlist, ie, the dictionary is partitioned into wordlists:

$$wordlist \,\mathrm{partition}\, dict$$

At any point in time, the dictionary has a search order associated with it. The search order is simply a sequence of wordlists that are to be searched:

$$search\_order : \mathrm{seq}\, wordlist$$

There is also the compilation wordlist:

$$compilation\_wl : wordlist$$

4.10 Defining words

When a new word is created, the system state is updated in three ways:

1. Its name is appended to the current compilation wordlist and thereby to the dictionary.
2. Its definition is appended to the *body* or *code* relations dependent on the type of word being defined.
3. The *immediate* relation is updated to indicate if the word is immediate or not.

Let us look at a few examples to see how this works.

4.10.1 High-Level words

We could define the word `+!` as:

```
: +! ( n addr -- ) DUP @ ROT + SWAP ! ;
```

This would add the name `+!` to the currently defined compilation wordlist:

$$compilation\_wl' = compilation\_wl \cup \{(244, (\text{“+!”}, 8270))\}$$

We now add the word's definition to the system. As it is a "high-level" definition we do this by adding an entry to the *body* relation:

$$body' = body \cup \{224 \mapsto \langle \texttt{DUP}, \texttt{@}, \texttt{ROT}, \texttt{+}, \texttt{SWAP}, \texttt{!} \rangle\}$$

Finally, we extend the *immediate* function so as to return a false value for this word:

$$immediate' = immediate \cup \{244 \mapsto \text{false}\}$$

4.10.2 Immediate words

The definition for the word `IF` could be:

```
: IF COMPILE ?BRANCH >MARK ; IMMEDIATE
```

This would add the name `IF` to the current compilation wordlist:

$$compilation\_wl' = compilation\_wl \cup \{(300, (\text{“}\texttt{IF}\text{”}, 10030))\}$$

The definition of the word is also added to the *body* relation:

$$body' = body \cup \{300 \mapsto \langle \texttt{COMPILE}, \texttt{?BRANCH}, \texttt{>MARK} \rangle\}$$

While the `IMMEDIATE` places a true mapping into the *immediate* function:

$$immediate' = immediate \cup \{300 \mapsto \text{true}\}$$

4.10.3 Code words

When a "code" level word, such as `ROT`, is defined, we add its name to the current compilation wordlist:

$$compilation\_wl' = compilation\_wl \cup \{(5, (\text{“}\texttt{ROT}\text{”}, 40))\}$$

We must now assume that its definition is correct and simply add the description of its function to the set of *axiom*atic definitions:

$$code' = code \cup \left\{5 \mapsto [\langle pstack(3), pstack(1), pstack(2) \rangle \frown tail\, tail\, tail\, pstack\,]\right\}$$

Finally, the *immediate* function is updated in the same manner as for "high-level" definitions. It is assumed that the word is not immediate unless the `IMMEDIATE` word is placed after its definition.

$$immediate' = immediate \cup \{5 \mapsto \text{false}\}$$

4.11 Dictionary Searching

In order to model the dictionary search, we define a boolean function that returns a true if a given word is in a given wordlist, otherwise it returns a false:

$$inwordlist_1(n, wl) = n \in \mathrm{dom}(\mathrm{ran}(wl))$$

A true result is obtained if n belongs to the set $\mathrm{dom}(\mathrm{ran}(wl))$. Let us clarify this by means of an example:

$$\begin{array}{rl} \text{Assume} & wl = \{(3, (\text{“}\texttt{SWAP}\text{”}, 30)), (4, (\text{“}\texttt{DROP}\text{”}, 36)), (5, (\text{“}\texttt{ROT}\text{”}, 40))\} \\ \text{then} & \mathrm{ran}\ wl = \{(\text{“}\texttt{SWAP}\text{”}, 30), (\text{“}\texttt{DROP}\text{”}, 36), (\text{“}\texttt{ROT}\text{”}, 40)\} \\ \therefore & \mathrm{dom}\,\mathrm{ran}\ wl = \{\text{“}\texttt{SWAP}\text{”}, \text{“}\texttt{DROP}\text{”}, \text{“}\texttt{ROT}\text{”}\} \end{array}$$

We can now define a function to find a given name within a given wordlist:

$$\begin{array}{rrl} find_1(n, wl) = & \mathsf{if} & n = first(second(wl)) \\ & \mathsf{then} & second(second(wl)) \\ & \mathsf{else} & find_1(n, front\ wl) \end{array}$$

This recursive definition says that if the name being searched for (n) in wordlist (wl) is the last name in the wordlist ($first(second(wl))$) then return its associated address ($second(second(wl))$). Otherwise, it repeats the operation on a new wordlist being the *front* of the current wordlist. Note that the definition for $find_1$ does not indicate what will happen if the name is not in the wordlist.

We now introduce a boolean variable to indicate if the word has been found or not:

$$wordfound : \{\mathrm{true},\ \mathrm{false}\}$$

We can now complete our model of the dictionary search operation by defining a function that takes a name and a search order as arguments, returning an address:

$$\begin{array}{rrl} find(n, so) = & \mathsf{if} & so \neq \varnothing \\ & \mathsf{then} & \quad \mathsf{if} \quad inwordlist_1(n, head\ so) \\ & & \mathsf{then} \quad find_1(n, head\ so) \\ & & \qquad\quad wordfound' = \mathrm{true} \\ & & \mathsf{else} \quad find(n, tail\ so) \\ & \mathsf{else} & wordfound' = \mathrm{false} \\ & & 0 \end{array}$$

This function works by checking that the required word (n) can be found in the first wordlist of the search order (so). If it can, we use the function $find_1$ to find it and set the variable *wordfound* to true otherwise we start again using the first wordlist from the remaining wordlists in the search order. Note that if the search order becomes empty, then we have searched through all of the given wordlists without finding the word. Hence, we simply set the variable *wordfound* to false. Thus, we can use the value of *wordfound* to indicate that the word has been found in one of the given wordlists.

Chapter 5

Stack Optimisation

In the previous chapter we saw how representing the FORTH stack as a sequence of untyped elements can be useful when formally defining the operations of a FORTH system, compiler or application.

This concept has lead us on to viewing the stack as a simple sequence of (untyped) elements. In turn, this has lead to the development of a compiler optimisation technique that uses these ideas. In this chapter, we review the traditional methods of optimisation used in native code FORTH implementations. We then go on to describe our new technique and how it can be applied to a micro-processor with a large register file, such as the Motorola 68000 or a RISC processor.

5.1 Introduction

Over the years, FORTH compilers have been implemented in a number of different ways. These include:

Threaded Code: In this type of system the compiler will simply generate a list of addresses for a definition. An "Inner Interpreter" is used to pass through the words definition. The first cell of the definition is the address of some assembler code that is able to interpret the rest of the definition. Every FORTH word in the dictionary is defined in this manner including assembler level words.

This is probably the most commonly used method of implementation. It is the implementation method promoted by the early standards (Forth Interest Group 1980; Forth Interest Group 1983). It leads to a system that tend to be small in size (as most of the definitions are lists of addresses) and have a very fast compile time. The method does lead to relatively slow execution timings. It is fairly simple to develop interactive debuggers and structured de-compilers (Buege 1984; Sjolander 1987; Bradley 1985) for such systems.

This method is known as "*Indirect threaded code*". A slight variation on this system (known as "*Direct threaded code*") is also widely used. In this variant, each word is assumed to be an assembler definition, thus when a word is invoked, the system makes a machine level call to the word. Such a system will compile a call to the inner interpreter as the first operation of a high-level definition. This method is well suited for systems that are capable of easily supporting two (or more) stacks.

The "*Direct*" system has a level of indirection removed, thus it tends to be slightly faster than "*Indirect*" systems. Applications consisting mainly of low level (assembler) definitions tend to show an execution speed increase due to the inner interpreter being invoked only when required. The size of the final code is dependent on the system and the complexity of the application. On a Z80 based system the code will be slightly larger than that produced with a *Indirect* system, whilst on a RISC based system there will be no size difference. If the application consists mainly of low level definitions the code may be smaller.

In general, it is considered that the slow execution speed disadvantage is overshadowed by the size and de-compiler advantages. The portability of high level definitions is also considered a major advantage of this method.

Subroutines: In some systems, a high level definition consists of a sequence of subroutine calls to the relevant words. Every word is assumed to be a machine code definition. This kind of system is referred to as using the "*Threaded Subroutine*" method as it is close to the *Direct Threaded Code* method using subroutines rather than an inner interpreter.

The main advantage in using this method is that it does away with the need for an inner interpreter. A result of this is a speed increase over *Direct Threaded* systems, although compilation speed may be slightly reduced (Dowling 1981).

The size of the code is dependent on the system and complexity of the application. On a 68000 system although we gain a speed advantage, the code produced is normally larger than *Direct Threaded Code*. On a RISC system there is not only a speed advantage but also a small saving in code size as we don't need to invoke an inner interpreter to interpret high level definitions.

This kind of coding is only useful on systems that can support two or more stacks. On systems that only support one (processor) stack, it is necessary to synthesize one of FORTH's stacks. It is normal to synthesize the "return" stack leaving the built in (faster) stack for the "parameter" passing.

It is a fairly simple step to make some of the low level kernel words macros, such that, rather than compiling a subroutine call, it compiles the instructions needed to perform the function directly. This is referred to as "*inline code*".

Typically, the code compiled into the definition consists of simple stack manipulations and some flow control operations. The code that is compiled "inline" in this way is normally fixed by the compiler designer. A number of people have looked at automating this process (Pawley 1984; Rose 1986; Almy 1987). In this chapter, we look into these methods and show how treating the stack formally (as a sequence of items) can lead to an advanced optimisation technique.

In such a system, we would expect to find a dramatic speed increase although a slight compilation speed decrease is also possible. We would also expect to see less use of the return stack. The compiler designer will have to make the choice of possible smaller slower code or slightly larger and faster code. Although the final code may be faster, it may also be larger, dependent on the compiler.

Hardware: A number of attempts have been made to produce micro-processors that are capable of executing a FORTH-like machine language. Chips such as the Novix–4016 (Golden, Moore, and Brodie 1985; Jennings 1985; Miller 1987), Harris RTX–2000 (Danile and Malinowski 1987; Jones, Malinowski, and Zepp 1987; Harris Semiconductor 1988c) and the MuP20 (Moore 1990a) have been designed along these lines. The binary instruction set of these processors is a basic implementation of the low level kernel required to execute a FORTH program. A compiler will generate native code to execute on one of these chips. As this code is close to the FORTH system, such compilers produce fast and small code.

A number of more advanced designs are currently under development including designs by Chuck Moore (Computer Cowboys), Marty Fraeman (Johns Hopkins University/Applied Physics Laboratory), Pedro Luis Prospero Sanchez (Cidade University in Sao Paulo) and Sergei Baranoff (St. Petersburg Institute for Informatics and Automation).

The majority of FORTH system use "*threaded code*" techniques, thus the compilers are quick and simple. The alternative "*native code*" techniques tend to use more space but offer more opportunity for optimisation. The rest of this chapter looks at some of these optimisation techniques and introduces a new one based on a formal view of the parameter stack.

It is worth noting that the nature of the language makes many of the traditional optimisation techniques (such as common subexpression, copy propagation, loop optimisation and code motion) irrelevant. There are, however, a couple of the traditional techniques (such as dead-code elimination and flow graphs) that can be used (Aho, Sethi, and Ullman 1986; Bruno and Lassagne 1975).

5.2 Code Generation

In a FORTH system that generates native code instead of threaded code, the operation of a word could simply be described by a sequence of subroutine calls to the relevant operations. This has been termed "*subroutine threaded code*". For example, a definition of the standard word `+!` could be:

```
: +!     ( n1 addr --   ; Add n to contents of addr )
  DUP    ( n1 addr addr )
  @      ( n1 addr n2   )
  ROT    ( addr n2 n1   )
  +      ( addr n1+n2   )
  SWAP   ( n2+n1 addr   )
  !      ( )
;
```

This would produce a operation body of:

```
JSR    DUP         Dup
JSR    _Fetch      @
JSR    ROT         Rot
JSR    _Add        +
JSR    SWAP        Swap
JSR    _Store      !
RTS                ; (Return to caller)
```

5.3 Inline Compilation

The code generated by this definition is just as simple to produce as in the threaded code technique. However, we now incur an overhead in using the subroutine (`JSR`) instruction. This places an inherent slowness into the system, a subroutine call has to place its return address on the stack, execute the target code and return to its calling location. On the 68000, the threaded code method is more complex and slightly slower (Almy 1987) whilst a RISC is additionally required to clear out its instruction pipeline. Anthony Rose (1986) has proposed a system of "inline compilation" that reduces this overhead.

The basic principal behind Rose's idea is that each word is to be compiled so that it can be used as a subroutine call (as with subroutine threaded systems). The length of the code (excluding the subroutine return (`RTS`) instruction) will be stored in the word's header. When a reference to the word is to be compiled, the compiler will compare the length of the word's body to a compilation variable (`CSIZE`). If the word is smaller in length to the given size then the code will be copied directly into the new definition. If, however, the code is over the limit set by `CSIZE`, a subroutine call to the code will be compiled into the new definition. By having a reasonable value for the "inline" limit (Rose suggest 13 for a 68000 based system), the overhead of making a subroutine call can be removed as the code is compiled directly into the definition. Note, this does not eliminate the subroutine calls to pre-compiled code, it simply reduces the frequency of them.

Rose also noted a requirement for two additional compilation flags, *subroutine* (`SUBR`) and *inline* (`INLINE`) to be used in the same manner as `IMMEDIATE`. If the `INLINE` flag is set, the word is compiled directly into the code ignoring the value of `CSIZE`. Conversely, if the `SUBR` flag is set, a subroutine call to the code will *always* be compiled.

5.4 Peep-Hole Optimisation

Using inline compilation, the word `+!` would now have a operation body of (in standard 68000 assembler notation):

```
MOVE.L   (A6),-(A6)        Dup

MOVEA.L  (A6),A0           @
MOVE.L   (A0),(A6)

JSR      ROT               Rot

MOVE.L   (A6)+,D0          +
ADD.L    D0,(A6)

MOVE.L   (A6)+,D0          Swap
MOVE.L   (A6),D1
MOVE.L   D0,(A6)
MOVE.L   D1,-(A6)

MOVEA.L  (A6)+,A0          !
MOVE.L   (A6)+,(A0)

RTS                        ;
```

Notice the additional time incurred by the passing of arguments on the stack which is then used by the next word in the definition. The `DUP` is used so that the `@` will not destroy the address on the stack. This is an additional overhead caused by using this inline compilation. However, if the word `@` were to be `IMMEDIATE`, it can scan back through the code just compiled. This means that it can recognise that a `DUP` was compiled just prior to it. Having done this it can now overwrite the code for `DUP` with some code that would perform the fetch without removing the address from the stack (or compile an implied `DUP@`). Given that all the basic[1] words perform this kind of scanning back to the previous word compiled, the body for `+!` would now be:

```
*                          Dup

   MOVEA.L   (A6),A0       @
   MOVE.L    (A0),-(A6)

   JSR       ROT           Rot

   MOVE.L    (A6)+,D0      +
   ADD.L     D0,(A6)

*                          Swap

   MOVEA.L   4(A6),A0      !
   MOVE.L    (A6)+,(A0)
   ADDQ      #8,A6

   RTS                     ;
```

The `!` word has replaced the code generated by `SWAP`. Notice how `!` has used the instruction `ADDQ #8,A6` to move the parameter stack back to its expected location.

Here a saving is made by each word scanning back to the previous word compiled. It is common for a word to be able to scan back up to four previously compiled words. If the word can scan back further (to the beginning of the definition), a greater saving can be achieved. The body for this could then be:

[1] There is an ongoing argument in the FORTH community as to what precisely constitutes this basic set of words.

```
MOVEA.L   (A6)+,A0     Dup @
MOVE.L    (A6)+,D0     Rot
ADD.L     D0,(A0)      + Swap !
RTS                    ;
```

This form of backwards scanning of the compiled code (known as "*peep-hole optimisation*" (Tanenbaum, van Staveren, and Stevenson 1982)) can give rise to some rather more optimal coding that would not otherwise be available. Any function that relies on information from the stack can be optimised in this way.

5.4.1 Conditionals

Another place for excessive optimisation is the conditional words (such as `IF`). Including all of the words that leave a boolean flag on the stack (such as `=`). Currently, the word `IF` simply interrogates the boolean flag on the top of the stack to make a conditional jump. If it were to scan backwards then the boolean flag would not be required as it could be integrated into the jump instruction at the condition test. For example, let us take the code "`0 > IF`", this would normally be compiled as:

```
        MOVE.L    #0,-(A6)       0

        MOVE.L    (A6)+,D0       >
        MOVE.L    (A6),D1
        CLR.L     D2             0 (false)
        CMP       D1,D0
        BLE       f0
        SUBQ      #1,D2          -1 (true)
0:      MOVE.L    D2,(A6)

        MOVE.L    (A6)+,D0       IF
        BEQ       <n>
```

Where `<n>` is the conditional offset. It is initially set to `0` while its correct value is calculated by the word '`THEN`'.

With the `>` and `IF` words producing optimal code, this could now be compiled as:

```
CLR.L     D0             0
MOVE.L    (A6)+,D1       >
CMP       D1,D0
BLE       <n>
```

Here the `>` word replaced the code to place a literal `0` on the stack as it is not required. The `IF` word replaces the processing (and production) of the boolean flag by placing its offset in the code produced by `>` (the `BLE` instruction), thus removing the need to pass the boolean flag from the one word to the next.

If multiple conditions are being tested (ie a logical `OR` or `AND` is being used), it would be possible to use lazy evaluation (Minker and Minker 1980; Hanson 1980) to reduce the time taken to evaluate the logical expression prior to the `IF`.

5.5 Registers

Some implementors have extended this thinking so that they place the top of the stack in a specific (*static*) register (Bradley and Saari 1988). Others have tried placing the top three items in registers. The second one being in an *Address* register rather than an *Data* register (`comp.lang.forth` 1992).

This form of optimisation can be used with all of the implementation methods. However, it is felt that the use of *static* registers to hold elements of the stack is cumbersome. The speed advantage is outweighed by the complexity of the system. Indeed this is often quoted as being the most frequent source of error in such systems (`comp.lang.forth` 1992).

5.6 Optimisation using a Stack image

In the rest of this chapter we present a new technique which allows the top items of the stack to be stored in internal registers. The registers used to store the items (S_0, S_1 and S_2) are to be allocated *dynamically*. In order to do this, the compiler will have to maintain a record of which register holds which stack element. This record is in the form of an image. To show this "*stack image*" we use a notation that relates a register to its stack element, thus the sequence:

$$< D_0, D_1, D_2 >$$

indicates that the top item of the stack (S_0) is stored in the register D_0 with the second stack element (S_1) in D_1 and the third (S_2) in D_2. Any stack item not given in the sequence is assumed to be on the physical stack of the micro-processor. Hence, the notation $< D_2, D_0 >$ signifies that S_0 is in register D_2, S_1 is in D_0 and that S_2 is not being held in an internal register but is on the micro-processors 'physical' stack.

Underflow of the parameter stack may still occur, however it is possible to include some form of checking into the compiler to check for this (Hoffmann 1991). By having a formal list of arguments into and out-of a word, the compiler will be able to find out if underflow will result from the expected usage.

To show how our system works, let us look at the standard word `SWAP`. If the top two elements of the stack are stored in registers, `SWAP` need only change the compilation stack image to achieve its run-time effect. However, if one or both of the elements are on the parameter stack then they will have to be brought into internal registers. We now have to update the stack image. In doing so it can also swap the items around.

Table 5.1 shows the possible compilation actions that can be taken by the word `SWAP` dependent on the current state of the stack image[2].

Code Generated:	Stack Image	
none	Before:	< D0, D1 >
	After:	< D1, D0 >
MOVE.L (A6)+,D1	Before:	< D0 >
	After:	< D1, D0 >
MOVE.L (A6)+,D1	Before:	<>
MOVE.L (A6)+,D0	After:	< D0, D1 >

Table 5.1: Example `SWAP` actions

5.6.1 Argument Passing

Sometimes it will be necessary to invoke these (compiling) words as a subroutine (Ie when invoked from the keyboard interpreter). In such cases we do not know the state of the stack. To counter this, we say that the stack image must always be

$$< \texttt{D0}, \texttt{D1}, \texttt{D2} >$$

[2] By this we mean the current values in the stack image.

on entry to and exit from the subroutine. Thus it is now the responsibility of the subroutine to make sure that the stack image is in the same form on exit of the routine.

Let us take the **SWAP** word. This now has two different actions to take depending on whether it was invoked as a subroutine or from the compiler. If the word is invoked from the compiler, it must compile the relevant code into the dictionary (as shown in table 5.1). Otherwise it will have to perform as a subroutine, the following code defines the action of the word when invoked as a subroutine:

```
_SWAP:   EXG      D1,D0
         RTS
```

This subroutine knows what the stack image is supposed to be on entry and exit of the code. By simply exchanging the contents of the top two registers, it has performed its function without having to alter the stack image.

The definition of the compiler word **SWAP** would have to be intelligent enough to discover whether it was invoked from the compiler or interpreter taking the required action. The following is a possible definition of the **SWAP** word[3]:

```
HEX
: SWAP  ( n1 n2 -- n2 n1  ; Swap over the top two stack elements )
  ?COMP IF
    ( In compilation mode - Take relevant action )
    SLEN @ CASE

     0 OF ( All argument on physical stack )
       IN-LINE   A6 )+  D0 .L MOVE
                 A6 )+  D1 .L MOVE  END-CODE
       , ,                              ( Place Op-Code into Definition )
       2 SLEN !                         ( Update Stack Len )
       0 >S0 1 >S1                      ( Set up Stack Image )
     ENDOF

     1 OF ( Second argument on physical stack )
       S0> 1+ 3 MOD DUP                 ( Find next register )
       IN-LINE   A6 )+ D0 .L MOVE  END-CODE
       200 * OR                         ( Replace D0 with correct reg.)
       ,                                ( Compile into the definition )
       2 SLEN !                         ( Update Stack Length )
       S0> >S1 >S0                      ( Update stack image )
     ENDOF

       ( Both in internal registers )
       S0> S1>                          ( Read old stack image )
       >S0 >S1                          ( Set new stack image )
     ENDCASE
  ELSE

    ( Called form Keyboard Interpreter )
    _SWAP

  THEN
; IMMEDIATE
```

Here we can see the definition of the kernel words, such as **SWAP**, are far more complex than the simple **CODE** definitions that we are used to in the more traditional compilers. However, only the basic (kernel) operations will need such a complex definition.

[3] The method of implementing the interrogation of the current interpreting/compiling state was chosen to show the concept, other methods may be better for full implementations.

In this definition, we have used many non-standard words. These have been used to make the definitions more readable. The function `?COMP` is used to detect whether the FORTH system is in interpret or compilation mode. The version of the word used here is different from its conventional usage in that it returns a flag rather than producing an error message. The words `S0>`, `S1>` and `S2>` read the value of the register holding the stack elements S_0, S_1 and S_2 respectively. In contrast, the words `>S0`, `>S1` and `>S2` place a register number in the stack image for the relevant element. The variable `SLEN` is used to hold the current size of the stack image. The action taken by the word at compile time is dependent of the size of the stack image.

Finally, we have introduced the new concept in the word `IN-LINE`. It is used to enter the assembler such that the machine instructions are compiled into the definition as literal values and not as executable code. This is the same as working out the op-codes by hand and coding them into the definition as literal values. However, `IN-LINE` uses the built in assembler to compute the op-codes for us, it also makes the source code a lot more readable.

Now let us apply this idea to our definition of `+!`. The body of the word (after compilation with such words) could now be:

```
*                              < D0,D1,D2 >
  MOVE.L   D2,-(A6)   Dup
  MOVE.L   D0,D2               < D2,D0,D1 >
  MOVEA.L  D2,A0      @
  MOVE.L   (A0),D2             < D2,D0,D1 >
*                     Rot      < D1,D2,D0 >
  ADD.L    D1,D2      +        < D2,D0 >
*                     Swap     < D0,D2 >
  MOVEA.L  D0,A0      !
  MOVE.L   D2,(A0)             <>
  MOVE.L   (A6)+,D0   ;
  MOVE.L   (A6)+,D1
  MOVE.L   (A6)+,D2
  RTS                          < D0,D1,D2 >
```

Notice how the ; word has compiled three `MOVE.L (A6)+,D`n instructions. This is in order to regenerate the correct stack image for exit from the subroutine. This is the "worst case" situation as all three registers have to be *pop*ped off the physical stack into registers.

5.6.2 Conditional execution

When a branch is made in the code (an `IF` instruction), the state of the stack image could differ between compile and run time. If the compiler did not take account of this it would compile code to work with one stack image when in fact another stack image is being used. To overcome this, we save a copy of the stack image at the start of the conditional (after the `IF`). At the end of the conditional (the word `THEN`), we will have to force the stack image to be the same as when the condition started (the saved image). In this way, we can say that the stack image will be the same if we skip over the conditional or if we execute it. Notice that the extra code to realign the stack is included with the condition. An example of this is shown in figure 5.1.

For an `IF...ELSE...THEN` construct, we can extend this idea such that the `ELSE` word will swap the current stack image with the saved one. This means that both parts of the conditional code start with the same stack image and that the *false* part is required to realign its stack to be the same as that at the end of the *true* part. Figure 5.2 shows an example of this method.

```
IF      ⟵ Save stack image
  true part
THEN    ⟵ Realign stack image
```

Figure 5.1: Handling a conditional

```
IF      ⟵ Save stack image
  true part
ELSE    ⟵ Swap current/saved stack images
  false part
THEN    ⟵ Realign stack image
```

Figure 5.2: Handling multiple conditions

5.6.3 Looping structures

It is possible to handle loop structures in the same way as conditional structures. We save the stack image at the start of the iteration and realign it at the end. Figure 5.3 shows how we would implement this idea on a `BEGIN`...`WHILE`...`REPEAT` structure.

```
BEGIN     ⟵ Save stack image
  condition test
WHILE     ⟵ Save stack image
  loop code
REPEAT    ⟵ Realign to BEGIN image
          ⟵ Recover WHILE stack image
```

Figure 5.3: Handling loops

This figure shows a peculiar problem with the looping system. When we compile the `BEGIN` word we do not know what kind of loop structure we are compiling. Thus, we save the current stack image in anticipation of it being used. All of the control structures that start with the word `BEGIN` will realign to this value at some point. The `WHILE` loop is a special case as we are required to remember the stack image twice, at the '`BEGIN`' and at the '`WHILE`'. The '`REPEAT`' command will realign the stack to the `BEGIN` stack image. On completion of the loop, the `WHILE` stack image will be in force, thus `REPEAT` will also have to reset the compilation stack image back to the `WHILE` version.

Chapter 6

The Cell Type

It is generally considered that the lack of typing in FORTH is useful. This can be seen by the definition of the stack to hold values of type "cell". The definition of the type *cell* is sufficiently vague to allow any data type. However, this can also be misleading and confusing. Here we present a theory that allows us not only to type the arguments of a function, but additionally to check that the arguments are correct for any given function.

6.1 Introduction

The ANS ASC X3/X3J14 Technical Committee defines a cell as:

> The primary unit of information in the architecture of a FORTH system. Data stack elements, return stack elements, addresses, and single-cell numbers are one cell wide. Cell size is implementation-defined, specified in integer address units and the corresponding number of bits. The size of a cell is an integral multiple of the size of a character.

Let us look at the following FORTH code:

```
X @ EXECUTE
```

where the variable `X` is holding an integer. The word `@` will fetch a value of storage class *cell* and place it on the stack. The word `EXECUTE` will then take the *cell* storage class and execute the related definition.

There are two types used in this example, "*integer*" and "*execution-token*". Both types belong to the storage unit class *cell*. In this example, we have the word `EXECUTE` expecting a value of type *execution-token* when there is a value of type *integer* on the stack. This is obviously a type clash. Due to the definition of a cell, we have no choice but to let this error stand. This is not a new problem, it has existed from the first implementations of FORTH.

6.2 Stack Types

One way to solve this problem is to implement some form of typing mechanism. Implementing a run time type checking mechanism would be too cumbersome to be of use. It would also restrict the programmer from performing certain "*tricks*" that require a change of type part way though a definition.

In this chapter, we propose a system that can be used to check the type requirements of a sequence of words at compile time. This has the advantage of not being operational at run time. It also has the advantage of not restricting the programmer from changing the type of a stack argument mid word.

This system can be used to check that any given program meets its stack requirements. This is not the same as saying that the program is complete or correct in operation. That is to say that a program does not invalidate the stack, but may be logically incorrect. This is the same as a Pascal program compiling, but not executing correctly. Such a program is known as having a "logic error" as opposed to a "syntax error" or a "type mismatch".

6.3 Notation

In order to discuss these ideas in a clear manner, we use the notations of set theory and new notations that we have developed for this system.

We give each word a "*type signature*" in the way that we currently give each word a signature (in comments). In order to make things look similar to the current practise, we use the notation (s_1 --- s_2) to indicate a words type signature. In this example, the word is expecting a type sequence s_1 on entry and will leave the type sequence s_2 on exit from the word.

It would be possible to define a word with a type signature of (a, b, c --- a, a) to indicate that the word will take three arguments of type a, b and c returning two values of type a on the stack. Using this system, it is be possible to prove that the sequence of words that makes up a new word will actually perform the required type transformation.

Let us take another example, this time we will use the word `SWAP`. This has a type signature of (w_1, w_2 --- w_2, w_1). Notice that here we are using the type w_1 to indicate a wildcard type, while the type w_2 indicates another wildcard type. Wildcards are items of unknown type.

We show a sequence of signatures by writing them next to each other. Thus a two word (signature) sequence would be written (s_1 --- s_2)(t_1 --- t_2) where s_1 is the stack image on entry to the sequence and t_2 is the stack image on exit from the sequence.

6.4 Rules

In order to discover if a sequence of type signatures perform the type transformation we require, we use a number of rules for manipulating the signatures. The rules are broken into three logical groups: composition; reduction and wildcard.

6.4.1 Composition Rules

The *Composition Rules* are used to rewrite two signatures into one new signature. We will use the notation (s_1 --- s_2)(t_1 --- t_2) to indicate two adjacent type signatures, where s_1, s_2, t_1 and t_2 are type sequences.

Rule 1: If s_2 is null (there are no types indicated) then we can add the requirements of the second word to that of the first, generating one signature.

$$\frac{(\ s_1\ \texttt{---}\ s_2\ )(\ t_1\ \texttt{---}\ t_2\ ), \#s_2 = 0}{(\ t_1, s_1\ \texttt{---}\ t_2\ )}$$

For example: (a, b ---)(c --- d) = (c, a, b --- d)

Here the first word takes arguments of type a and b off the stack and returns no arguments. The second word takes an argument of type c off the stack and returns a value of type d. Hence the argument c must be on the stack before this sequence is executed.

Rule 2: If t_1 is null (the second word takes no arguments) then we can append the results of the second word to those of the first word.

$$\frac{(\ s_1\ \texttt{---}\ s_2\ )(\ t_1\ \texttt{---}\ t_2\ ), \#t_1 = 0}{(\ s_1\ \texttt{---}\ s_2, t_2\ )}$$

For example: $(\ a\ \text{---}\ b\ )(\ \ \text{---}\ c\ ) = (\ a\ \text{---}\ b, c\ )$

The second word takes no arguments and so the combination of the two sequences can be given by simply adding t_2 onto the end of s_2.

Rule 3: If the last element of s_2 does not match the last element of t_1 then we have a type clash.

$$\frac{(\ s_1\ \text{---}\ s_2\ )(\ t_1\ \text{---}\ t_2\ ), last\ s_2 \neq last\ t_1}{0}$$

Eg: $(\ a\ \text{---}\ a, b\ )(\ a, c\ \text{---}\ d\ ) = 0$

Here we have the first word leaving an element of type b on the stack while the second word requires an element of type c. This is a type clash and is written as 0.

6.4.2 Reduction Rules

The following *Reduction Rule* is used to reduce the type signatures until a composition rule can be used on the sequence.

Rule 4: If the last element of s_2 is the same as the last element of t_1 then the types do not clash and the argument passing is internal to the sequence of operations. Hence we can rewrite the sequence removing this element.

$$\frac{(\ s_1\ \text{---}\ s_2\ )(\ t_1\ \text{---}\ t_2\ ), last\ s_2 = last\ t_1}{(\ s_1\ \text{---}\ front\ s_2\ )(\ front\ t_1\ \text{---}\ t_2\ )}$$

Eg: $(\ a\ \text{---}\ b\ )(\ a, b\ \text{---}\ c\ ) = (\ a\ \text{---}\ \ )(\ a\ \text{---}\ c\ )$

The first word passes an argument of type b to the second word. This is internal to the sequence of operation and so does not need to be shown.

6.4.3 Wildcard Rules

The remaining rules are intended to provide for wildcards, where a wildcard argument is able to match with an argument of any known type. We refer to these as *wildcard rules* even though they are reducing the type signature and thus can be considered as reduction rules. We indicate a known type as being a member of the set $\mathbb{K}$ and a wildcard type as being a member of the set $\mathbb{W}$.

Rule 5: If the last element of s_2 is of a known type and the last element of t_1 is a wildcard we remove the matching items, rename any additional occurrences of the wildcard in the second signature with the known type from the first signature.

$$\frac{(\ s_1\ \text{---}\ s_2\ )(\ t_1\ \text{---}\ t_2\ ), last\ s_2 \in \mathbb{K}, last\ t_1 \in \mathbb{W}}{(\ s_1\ \text{---}\ front\ s_2\ )((\ front\ t_1\ \text{---}\ t_2\ )[last\ s_2 / last\ t_1])}$$

Example: $(\ a\ \text{---}\ b, c\ )(\ w_1, w_2\ \text{---}\ w_1, w_2, w_1\ )$
$\Rightarrow\ (\ a\ \text{---}\ b\ )((\ w_1\ \text{---}\ w_1, w_2, w_1\ )[c/w_2])$
$\Rightarrow\ (\ a\ \text{---}\ b\ )(\ w_1\ \text{---}\ w_1, c, w_1\ )$

The first word passes the second word an argument of type c which is matched with the wildcard w_2 expected by the second word. Thus we can determine the type of w_2 for the second signature.

Rule 6: If the last element of s_2 is a wildcard and the last element of t_1 is of a known type, we can remove the matching types and replace any occurrences of the wildcards in the first signature by the known type.

$$\frac{(\ s_1\ \text{---}\ s_2\ )(\ t_1\ \text{---}\ t_2\ ), last\ s_2 \in \mathbb{W}, last\ t_1 \in \mathbb{K}}{((\ s_1\ \text{---}\ front\ s_2\ )[last\ t_1 / last\ s_2])(\ front\ t_1\ \text{---}\ t_2\ )}$$

For Example: $(\ w_1, w_2\ \texttt{---}\ w_2, w_1\ )(\ a, b\ \texttt{---}\ c\ )$
$\Rightarrow\quad ((\ w_1, w_2\ \texttt{---}\ w_2\ )[b/w_1])(\ a\ \texttt{---}\ c\ )$
$\Rightarrow\quad (\ b, w_2\ \texttt{---}\ w_2\ )(\ a\ \texttt{---}\ c\ )$

We can determine the type of w_1 because it must match the type b given in the second word.

Rule 7: If there are wildcard types in the first signature and similarly named wildcard types in the second signature, we rename the wildcards in the second signature by decorating them with a prime.

$$\frac{(\ s_1\ \texttt{---}\ s_2\ )(\ t_1\ \texttt{---}\ t_2\ ), \mathrm{ran}(s_1 \cup s_2) \cap \mathrm{ran}(t_1 \cup t_2) \cap \mathbb{W} \neq \varnothing}{(\ s_1\ \texttt{---}\ s_2\ )((\ t_1\ \texttt{---}\ t_2\ )[w'/w])}$$

Eg: $(\ w_1, w_2\ \texttt{---}\ w_2, w_1\ )(\ w_1, w_2\ \texttt{---}\ w_2, w_1\ )$
$\Rightarrow\quad (\ w_1, w_2\ \texttt{---}\ w_2, w_1\ )((\ w_1, w_2\ \texttt{---}\ w_2, w_1\ )[w'/w])$
$\Rightarrow\quad (\ w_1, w_2\ \texttt{---}\ w_2, w_1\ )(\ w_1', w_2'\ \texttt{---}\ w_2', w_1'\ )$

We have renamed all of the wildcards in the second signature to be different to those in the first signature.

Rule 8: If the last element of s_2 is a wildcard and the last element of t_1 is a wildcard, we can remove the matching wildcards, renaming all remaining occurrences of the wildcard in the second signature with the wildcard from the first signature, provided that the wildcard does not already exist in the second signature (there is not a name clash).

$$\frac{(\ s_1\ \texttt{---}\ s_2\ )(\ t_1\ \texttt{---}\ t_2\ ), \mathit{last}\, s_2 \in \mathbb{W}, \mathit{last}\, t_1 \in \mathbb{W}, \mathit{last}\, s_2 \notin \mathrm{ran}(t_1 \cup t_2)}{(\ s_1\ \texttt{---}\ \mathit{front}\, s_2\ )((\ \mathit{front}\, t_1\ \texttt{---}\ t_2\ )[\mathit{last}\, s_2/\ \mathit{last}\, t_1])}$$

Example: $(\ w_1, w_2\ \texttt{---}\ w_2, w_1\ )(\ w_1', w_2'\ \texttt{---}\ w_2', w_1'\ )$
$\Rightarrow\quad (\ w_1, w_2\ \texttt{---}\ w_2\ )((\ w_1'\ \texttt{---}\ w_2', w_1'\ )[w_1/w_2'])$
$\Rightarrow\quad (\ w_1, w_2\ \texttt{---}\ w_2\ )(\ w_1'\ \texttt{---}\ w_1, w_1'\ )$

The wildcard w_1 from the first signature has been matched with the wildcard w_2' from the second signature. The operation of this rule is exactly the same as rule 4 with the exception that it is for wildcard arguments and not for arguments of known types.

6.5 Simple Examples

Here are some examples of how you would compose two or more signatures together using these rules.

1.

$(\ a\ \texttt{---}\ b, c, d\ )(\ w_1, w_2\ \texttt{---}\ w_2, w_1\ )$	
$(\ a\ \texttt{---}\ b, c\ )(\ w_1\ \texttt{---}\ d, w_1\ )$	Resolve wildcard (5)
$(\ a\ \texttt{---}\ b\ )(\ \texttt{---}\ d, c\ )$	Resolve wildcard (5)
$(\ a\ \texttt{---}\ b, d, c\ )$	Combine (2)

2.

$(\ w_1, w_2, w_3\ \texttt{---}\ w_2, w_3, w_1\ )(\ a, b\ \texttt{---}\ c\ )$	
$(\ b, w_2, w_3\ \texttt{---}\ w_2, w_3\ )(\ a\ \texttt{---}\ c\ )$	Resolve wildcard (6)
$(\ b, w_2, a\ \texttt{---}\ w_2\ )(\ \texttt{---}\ c\ )$	Resolve wildcard (6)
$(\ b, w_2, a\ \texttt{---}\ w_2, c\ )$	Combine (2)

Since the naming of wildcards is arbitrary we could simply write the last line of this example as $(\ b, w, a\ \texttt{---}\ w, c\ )$.

3.

(w_1, w_2 --- w_1, w_2, w_1)(w_1, w_2 --- w_1, w_2, w_1)	
(w_1, w_2 --- w_1, w_2, w_1)(w_1', w_2' --- w_1', w_2', w_1')	Rename (7)
(w_1, w_2 --- w_1, w_2)(w_1' --- w_1', w_1, w_1')	Match wildcards (8)
(w_1, w_2 --- w_1)(--- w_2, w_1, w_2)	Match wildcards (8)
(w_1, w_2 --- w_1, w_2, w_1, w_2)	Combine (2)

4. Let us assume the following signatures for FORTH words:

```
DROP  ( w --- )
OVER  ( w1, w2 --- w1, w2, w1 )
SWAP  ( w1, w2 --- w2, w1 )
 ROT  ( w1, w2, w3 --- w2, w3, w1 )
```

We can show that the sequence `OVER ROT DROP` has the same type signature as the word `SWAP`:

(w_1, w_2 --- w_1, w_2, w_1)(w_1, w_2, w_3 --- w_2, w_3, w_1)(w ---)	
(w_1, w_2 --- w_1, w_2, w_1)(w_1', w_2', w_3' --- w_2', w_3', w_1')(w ---)	(7)
(w_1, w_2 --- w_1, w_2)(w_1', w_2' --- w_2', w_1, w_1')(w ---)	(8)
(w_1, w_2 --- w_1)(w_1' --- w_2, w_1, w_1')(w ---)	(8)
(w_1, w_2 ---)(--- w_2, w_1, w_1)(w ---)	(8)
(w_1, w_2 --- w_2, w_1, w_1)(w ---)	(2)
(w_1, w_2 --- w_2, w_1)(---)	(8)
(w_1, w_2 --- w_2, w_1)	(2)

6.6 Multiple Signatures

It is possible for a FORTH word to have more than one acceptable signature. Indeed there are many words in FORTH that require more than one signature. For this reason we have introduced the "+" symbol to indicate the existence of another possible signature for the same word.

Let us take the FORTH word **AND**, there are two functions associated with this word. The first is that of a logical (Boolean) **AND**, while the second is that of a binary (bitwise) **AND**. The signature for a Boolean **AND** is (*flag*, *flag* --- *flag*), while the signature for a bitwise **AND** is (*logical*, *logical* --- *logical*), thus the true signature is:

$$\text{sig}(\texttt{AND}) = (\ \mathit{flag}, \mathit{flag}\ \texttt{---}\ \mathit{flag}\ ) + (\ \mathit{logical}, \mathit{logical}\ \texttt{---}\ \mathit{logical}\ )$$

The correct signature will be used in composition due to the naming of a known type. Let us assume that the FORTH word **IF** has the signature (*flag* ---). When we come to compose the sequence **AND IF** we will know (from the signature of **IF**) that the Boolean **AND** signature is required.

Notice that we have also introduced the notation sig(α) to indicate all of the possible signature compositions of the phrase α.

6.7 Pass by reference

We indicate a pointer to a known type by writing $*^n k$. Where the $*^n$ is used to indicate n levels of indirection and the k is the known type being referenced. For simplicity we write $*k$ to indicate $*^1 k$. The notation $*^0 k$ is the same as the basic type k without indirection.

A possible definition of the FORTH word `@` would be ($*w$ --- w), however we have not defined the pointer type to be able to point to wildcard types. Hence the actual signature for `@` is:

$$\sum_{k \in \mathbb{K}} (\ *k\ \texttt{---}\ k\ )$$

This produces a collection of signatures, (one for every entry in $\mathbb{K}$). The correct signature will be selected when this word is being composed.

6.8 Control Structures

We use the ideas of multiple signatures (and summation) to show all of the possible paths through a control sequence. This is best shown by example.

Let us take the FORTH statement: `IF` α `ELSE` β `THEN`. We must compose the signature for both cases of the `IF` condition. Hence for a true condition the sequence (*flag* ---)sig(α) exists, while for a false condition the sequence (*flag* ---)sig(β) exists. These two signature can be written as one multiple signature:

$$(\ \mathit{flag}\ \texttt{---}\ )\,(\mathrm{sig}\,(\alpha) + \mathrm{sig}\,(\beta))$$

For a more complex control structure, such as `BEGIN` α `WHILE` β `REPEAT`, we have no way of knowing how many times the loop will be executed. We must therefore produce a multiple type signature for all the possible different number of iterations:

$$\sum_{i=0}^{\infty} (\mathrm{sig}\,(\alpha)\ (\ \mathit{flag}\ \texttt{---}\ )\mathrm{sig}\,(\beta))^{i}\ \mathrm{sig}\,(\alpha)\ (\ \mathit{flag}\ \texttt{---}\ )$$

However, it is normally the case that a loop of this form is "balanced" in terms of its stack arguments. In the case of a balanced stack, the loop can be simplified to a single term. If the sequence sig(α)(*flag* ---)sig(β) can be reduced to a signature of (s --- s), (ie a balanced signature) we can reduce this signature to:

$$(\ s\ \texttt{---}\ s\ )\mathrm{sig}(\alpha)(\ \mathit{flag}\ \texttt{---}\ )$$

In order to fully satisfy ourselves that a program is complete, we must follow though every single path of execution. We can say that a word definition (or program) is type correct if its expected input and output types can be reduced to the single signature holding the same input and output types.

For example: Let us define a FORTH word `EXAMPLE` which takes an input signature of s and is expected to produce an output signature of t. The word is type correct if

$$\mathrm{sig}(\texttt{EXAMPLE}) = (\ s\ \texttt{---}\ t\ )$$

It should be noted that we are currently unable to check words that make use of the `EXECUTE` word. For example, given the definition:

```
: TEST ( char --- ) 'TEST @ EXECUTE ;
```

it would be possible for us to check that the body of `TEST` is correct. However the action of `TEST` is to execute the definition, the execution token of which, is stored in the variable `'TEST`. As we do not know the signature of this definition, we can not check against the specification given, ie (*char* ---).

This problem may be overcome by expanding the rôle of the execution token to include a type signature within a type signature. Ie, the signature (*char*, (*char* ---) ---) indicates a character and an execution token are expected where the execution token has the signature of (*char* ---). There are a number of ways in which one might resolve this restriction, several of which are currently under investigation.

6.9 Casting

There are occasions when a programmer will want to convert the type of a stack item that is not catered for by the default matching type signatures. We have introduced a notation that will allow the programmer to alter the current type signature at compile time.

Let us assume that the programmer would like to convert a single-cell integer into an execution token. He would have to add the following line to his code:

```
<< int --- token >>
```

Where the FORTH word `<<` enters into a "*alter type signature*" mode. He then gives a representation of what he expects the current stack type signature to be (`int`). The word `---` is used to indicate the end of the current stack and the start of a description indicating what he would like the current stack type signature to become (`token`). Finally, the word `>>` replaces the current type signature with the required signature.

Obviously, there are many checks we can make at this point. The number of stack items expected (between `<<` and `---`) must be equal to (or less than) the actual number of item calculated to be on the stack (and of the correct type). The number of stack items given in the expected part must be the same as the number given in the wanted part (between the `---` and `>>`), thus protecting the compilers image of the stack.

6.10 Strong vs Weak Typing

6.10.1 Strong Typing

In a strongly typed system, every variable will have a known type associated with it. Hence a single-cell variable that has been defined to hold an integer could not hold a token as that would lead to a type clash.

A strongly typed system would be difficult to implement compared to a weakly typed system. It would have to keep track of the type associated with each memory cell in the system where as a weakly typed system would not retain this information. Due to the nature of types in FORTH, a strongly typed system would require the programmer to give additional information. See sections 4.6 and 7.3.1 for a discussion on FORTH's type structure.

In a strongly type system, the programmer would have the benefit of peace of mind, insomuch as he knows that the system will report an error if he attempts to develop code that uses the stack in what would be considered the wrong way.

This can be seen by examining the following code:

```
X @ EXECUTE
```

This would have the following type signatures:

(--- *int*)(**int* --- *int*)(*token* ---)

This would obviously clash on the (**int* --- *int*)(*token* ---) section.

In order to compile this code the programmer would have to write:

```
X @ << int --- token >> EXECUTE
```

To convert the *int* returned from the `@` into the *token* that is excepted by `EXECUTE`. Thus the programmer has to explicitly instruct the compiler to make the conversion and allow this code.

6.10.2 Weak Typing

In a weakly typed, system all memory cells will be defined to hold any of the known types. This is simpler to implement, however it does not bring with it the same peace of mind that a strongly typed system would.

If we take the same code as before:

```
X @ EXECUTE
```

which will now have a type signature of:

(--- $*k$)($*token$ --- $token$)($token$ ---)

We can see that in the weakly typed system the `X` returns a referenced known type ($*k$) this will be matched with the referenced token ($*token$) type required by `@`. Thus, this code will be acceptable to a weakly typed system. Hence, a weakly typed system can aid in program construction but will not be able to catch misusage of variables.

Chapter 7

A FORTH Type Checker

In this chapter we look at the possibility of implementing a FORTH type checker based on the rules given in the previous chapter. We will be calling this type checking program "FLINT" by analogy with the `lint` program used for checking C programs.

FLINT is to provide a consistency check of FORTH source code. It will examine a file, checking high level FORTH definitions for stack depth and stack types. If an error is detected, the offending word, line, file and an appropriate error (or warning) message will be written to an error log. Checking will continue after the erroneous word.

In this chapter we give an initial specification for the FLINT program.

7.1 Invocation

The FLINT program should be used from the command line. The user will give the command "`flint foo`" to instruct the FLINT system to check the FORTH source code given in the file "`foo.fth`". The system should also check all of the "include" files used.

The user should have the ability to select warnings or errors only. The default being that both warnings and errors are written to the error log (in this instance "`foo.log`"). The user should be able to provide a command line switch to indicate the production the form of report they wish, possible switches are:

- `-E` Report errors only to the error log.
- `-W` Report warnings only to the error log.
- `-C` Check the file, reporting number of errors and warnings to the video. Does not produce an error log but simply counts the errors.
- `-V` Produce verbose error/warning reports.
- `-S` Produce additional statistical information at the end of the error log.
- `-O` Set maximum stack size (for overflow checking).

For such a system to be of any real practical use it must have the ability of being extended by the application code. As such, the system is to provide a basic programming ability that can be "hidden" from the FORTH compiler.

7.2 Stack Notation

For this system to operate a formal stack notation system must be provided. A system based on the following rules should be provided:

```
<stack-definition>  ::=  [<stack-items>] --- [<stack-items>]
    <stack-items>   ::=  <stack-item> [<stack-item>]
     <stack-item>   ::=  [<reference>] <type>
      <reference>   ::=  *[<reference>]
```

Thus the user is allowed to place as many type indicators before the `---` part (verbalised as "gives" or "giving") indicating what is currently on the stack. He may also place as many type indicators after the "gives" indicating what is left on the stack after the operation has been performed. The type indicator may consist of any number of references to a given type, where <*type*> is a known type (of a known type class) defined by the system (or by the user) or a wildcard. A reference to a type is the address of the type. Hence if we have a type "`u`" indicating an "unsigned integer" (of class "single-cell") then we indicate the "address of an unsigned integer" with the type indicator "`*u`". The type "`**u`" is the "address of an address of an unsigned integer" (this is the same as the $*^2 u$ notation used in section 6.7).

In this discussion we have referred to "user-defined types" and "classes of types". This relates back to the need to allow an application to expand on the types available in the checking mechanism.

7.3 Commands

The commands of the type checking mechanism should be totally "hidden" from the normal FORTH compiler. Hence all control operators must be given in comments:

```
<command>  ::=  \  <function> [ ;  <comment>]
             |  (  <function> [ ;  <comment>] )
```

Thus provided that the command <*function*> is the first word in a comment the function will be invoked. If FLINT does not recognise the text at the start of a comment to be a command, the comment is ignored and processing continues from the end of the comment. A FORTH system will simply ignore the comments, thus "hiding" the type commands from the FORTH compiler.

The following table lists the possible commands. The requirement for the command and its function is given in more detail in the following sections.

```
<function>  ::=  <type-command>
             |   <stack-command>
             |   <assume-command>
             |   <assert-command>
             |   <syntax-command>
```

7.3.1 Classes

There are to be a number of type classes. The user will not be able to provide any additional type classes[1] this must be coded into the FLINT system. The system will have the following classes:

Single-Cell: The base type for all items that occupy a single cell. There is provision for up to 255 pre- and user-defined types of this class.

Double-Cell: The class of types that require two cells on the stack. If any attempt is made to access part of the item then a type violation is reported. There is provision for up to 255 pre- and user-defined types of this class.

[1] This is a limitation of the proposed implementation method. It is hoped that a method of allowing the user to add new type classes may be addressed in the future.

Two-Cell: A class of types that may be considered a double number at times, but is really made up of two single-cell types. Such types use double number words (such as `2!` and `2@`) when accessing two single-cell items. There is provision for up to 255 pre- and user-defined types of this class.

Reference: A special class for address types. All locations must have a type associated with it. It is possible to make reference to all possible types[2] (no matter their class). Thus the reference type must be able to cater for all possible classes including a reference to a reference to a structure type. So, the reference class is really a super-class encompassing references to all types within it. FLINT should be able to cater for up to 255 levels of referencing (ie $*^{255}k$).

Wildcard: A special class of single-cell wildcards. An item of type wildcard will match with any other single-cell type. This is used in the definition of words such as `SWAP` where the definition is "`w1 w2 --- w2 w1`" (verbalised as "wild-one, wild-two, gives wild-two, wild-one"). `w1` and `w2` are considered to be two separate types of class wildcard. Thus when `w1` is matched to a known type, `w2` may be matched with another. There are 255 possible wildcard types under this class.

Double-Wildcard: A special class of double-cell wildcards. This operates in the same way as single-cell wildcards except that a double-cell wildcard will match with any value of the double-cell class, two-cell class or two values of the single-cell class. There are 255 possible wildcard types under this class.

7.3.2 Type Command

The type command allows the application programmer to add a new type to the list of known types. The type is defined to be of a given type-class. The format of the type command is:

```
<type-command>  ::=  type:  single-cell  <type>
                  |  type:  double-cell  <type>
                  |  type:  two-cell  <type>  =
                            <cell-item><cell-item>
                  |  type:  structure  <type>  =  <stack-items>
```

The `type:` command is followed by a one of four type-class identifiers. The name of the new type is then given with the remainder of the command dependent on the class identifier:

`single-cell` The new type will be defined to be of one cell in length. All of the base types are defined to be of the single-cell class.

`double-cell` The new type is defined to be two cells long. Any attempt to gain access to a part of the type will be considered a type violation. For such access an "assume" command will be required to convert the double cell into two single cells.

`two-cell` The new type is defined to be two cells long. The type name is followed by two <*cell-item*>s (`single-cell` type names) that comprise this double cell. Ie, to define a co-ordinate pair that may be broken down into its two (single cell) signed integer parts. Hence when the `two-cell` type is used the system considers the stack image to be of the two `single-cell` types. Thus allowing a convenient short hand notation for such "doubles".

`structure` The new type is a compound structure consisting of other stack items[3]. All the types must be defined before the structure is declared. A structure may only be passed by reference. Any attempt to access the structure will result in a type violation. When an item is to be extracted from the structure an "assume" command should be used[4].

[2] Pass the address of a type, no matter what the type.

[3] Including possible other structures and references.

[4] Given that we know the size and position of each element in the structure, it should be possible to validate an access into a structure. This extension has not, as yet, been investigated.

7.3.3 Stack Command

The stack command is used to give the expected stack operation of a word. Its format is:

<stack-command> ::= `stack` : *<stack-definition>*

The system will scan the definition of the next word to check that its definition meets the given stack. FLINT will assume the stack will be as given on entry and check that the stack is as given on exit. If the expected exit condition is not the same as the "found" condition, an error is reported.

When further words use the word, the "expected" condition will be used and the word will be flagged with a warning.

The given stack image will be placed into a buffer and will be associated with all new words until a new `stack:` command is given, a `;` is found or an "include" is found. Thus the following code fragments are all valid.

```
\ Stack: --- *n  | \ Stack: w --- w w | : xxx ( stack: w --- w w )
VARIABLE A       | : xxx              |   ...
VARIABLE B       | ... ;              | ;
```

It may be possible to do without the "`stack:`" part of this command and assume that all commands are "stack" unless their name is found. Thus, the format of the command would now read:

<stack-command> ::= [`stack` :] *<stack-definition>*

However, this would make it more difficult to write good comments.

7.3.4 Assume Command

The assume command will force the system to make an assumption about the current stack type. Its format is:

<assume-command> ::= `assume` : *<stack-items>* `---` *<stack-items>*

FLINT will first check that its current stack image is the same as the stack image given in the command, before the "gives". Only the top most elements need be given. It will then replace those elements with the ones given in the stack image after the "gives". An example usage would be to convert an integer into an execution token:

```
\ assume: int --- token
```

This allows the programmer to "cast" form one type to another without supplying code to perform the transformation. The user is allowed to transform any class into any class. The number of *<stack-items>* on either side of the "gives" may vary. Hence:

```
\ assume: w ---
```

is a valid assumption that removes a single-cell item from the stack. While:

```
\ assume: --- n
```

is also a valid assumption that introduces a signed integer to the stack.

7.3.5 Assert Command

The assert command instructs the system to check the current stack image against that given in the command. Its format is:

```
<assert-command>  ::=  assert : <stack-items>
                   |   check : <stack-items>
```

If the current stack image does not match that given in the command, an error is reported giving the current stack image. If the stack holds more items than is given in the assert command, only those items given will be checked.

It may be better to insist that the given stack must match all of the current stack items, thus forcing the programmer to identify the complete stack and not only the top-most elements they are interested in. This has the advantage that the programmer will realise how many items are on the stack. This could also be extended to include a special stack item of "...", as the first element to indicate an unknown number of elements before the stack description, thus providing the "partial" check we currently have.

7.3.6 Syntax Command

The syntax command provides the ability for the programmer to define additional syntax structures. Its format is:

```
<syntax-command>  ::=  syntax : <word><syntax-items>
   <syntax-items>  ::=  <syntax-item> [<syntax-item>]
    <syntax-item>  ::=  <<text>> <delimiter>
```

The syntax command is only required when defining a word that defines its own syntax. The following shows some FORTH words and their syntax definitions. Note the space delimiter (␣) on the `CREATE` definition.

```
CREATE   \ syntax: CREATE <word>␣
ABORT"   \ syntax: ABORT" <error message>"
```

This facility is provided so that, when the new <*word*> is used, the system can take account of the fact that the word will scan ahead in the input stream (as far as the indicated <*delimiter*>). The given syntax is associated with the next or currently defined word. Thus the following two code fragments will give the same results:

```
\ stack: ---                        | : mess ( stack: --- )
\ syntax: mess <id>  <message>#     | ( syntax: mess <id> <message># )
: mess ... ;                        | ... ;
```

It should be possible for FLINT to check that any word defined as having a syntax does have the given syntax. Hence any word that includes a syntax word (ie, uses a word with a syntax) but does not declare a syntax, could generate a warning. If there is a declared syntax and the "found" syntax does not match then an error may be reported.

7.4 Variable Stack Items

7.4.1 Or — |

There are words that would benefit from having more than one possible stack image. There are occasions where you would not want to place a wildcard, however, the value may be of two (or more)

types. Thus we introduce the "|" notation to indicate a possible other valid type for a given stack entry. Hence, to indicate that a word takes a single item from the stack, being an unsigned integer (`u`) or a signed integer (`n`), the definition would be:

```
u | n ---
```

Note, that the | is considered to be left associative. Hence:

```
n u | n | f --- n | u f
```

indicates that the word takes two value from the stack, the first being a signed integer (of type `n`) and the second being an unsigned integer (`u`) or a signed integer (`n`) or a boolean flag (`f`). The result of the function is either a singed integer (`n`) or an unsigned integer (`u`), and a flag (`f`).

FLINT will treat such an entry as a special (restricted) form of wildcard. When the wildcard is matched with a known type, it becomes that type for the rest of the definition.

We must redefine <*stack-image*> thus:

<*stack-item*> ::= [<*reference-part*>] <*type*> [| <*stack-item*>]

7.4.2 Alternative descriptions — +

There are words that have a stack image that do not differ in type but in the number of arguments. One such word is `?DUP`, for this we introduce the "`+`" notation. A stack definition for the word `?DUP` could be:

```
u --- u + u --- u u
```

Indicating that the word takes an unsigned value (`u`) and returns an unsigned value, or it takes an unsigned value and returns two unsigned values.

On words such as these, FLINT will match the relative stack description dependent on the matching types. When such a word is used in a definition, the system will pass though that definition twice, using different versions of the stack definition.

This form of programming is not recommended, thus the use of such words will produce a warning and all words that are defined using it will be flagged with a warning.

Thus we must redefine <*stack-definition*>:

<*stack-definition*> ::= [<*stack-items*>] `---` [<*stack-items*>]
[`+` <*stack-definition*>]

It should be noted that the "|" notation can be thought of as a special form of the "`+`" notation. Ie, that stack definition:

```
n --- n | f
```

can be thought of as being the same as the stack definition:

```
n --- n + n --- f
```

7.5 Flow Control

FLINT is able to handle the basic flow control words. There is no mechanism for extending this system to cater for application defined control mechanisms. However this could be added by taking note of where the flow control words are used. This will be made simpler by the adoption of the ANSI standard for the writing of new control words.

The following is a list of flow control words and the action take by FLINT when the word is encountered:

`RECURSE` will compare the current stack image to the entry stack assumption. If there is a type mismatch an error is reported. Note: if additional items are left on the stack, this is **not** considered an error.

`?DUP` is defined as "`u --- false + u --- u true`" thus the code will be checked twice, once for the "`true`" condition, once for the "`false`" condition.

`IF` ... is defined as "`flag ---` " (where "`flag`" is defined as "`true | false`"). It will consume the flag and save the current stack image in a buffer.

...`ELSE` ... will swap the contents of the saved buffer with the current stack image.

...`THEN` compares the current stack image with the one in the saved buffer. Thus, checking that the stack image has not been changed by the `IF` condition. This also checks that an `IF`...`ELSE`...`THEN` statement leaves the stack in the same condition, no matter which execution path is taken. An error is produced if these stack images do not match.

`BEGIN` ... will save the current stack image in a buffer for later processing.

...`AGAIN` compares the current stack image to the buffered one. If they do not match exactly an error is produced. Thus a `BEGIN`...`AGAIN` sequence must have a balanced stack.

...`UNTIL` consumes a flag and then compares the current stack image with the buffered one. If they do not match an error is given.

...`WHILE` ... will consume a flag, then compares the current stack image against the buffered image. An error exists if they do not match.

...`REPEAT` compares the current stack image against the buffered one, reporting an error if they differ. Hence the stack image must be the same at `BEGIN`, `WHILE` (except for the flag) and `REPEAT`. Thus, no matter how many times the loop is executed we know the condition of the stack on exit.

`DO` ... will consume two (signed) numbers, then save the current stack image.

...`LEAVE` ..., definition take from (ANSI 1991), compares the current stack image against the buffered image that will take effect if the leave were to be executed. An error exists if there is a mismatch.

...`LOOP` checks that the current stack image matches with the buffered one. An error exists if not. Note that this enforces "balanced" stacks when using the `DO`...`LOOP` structure.

...`+LOOP` is the same as `LOOP` except it first consumes a signed integer.

Note: If at any comparison too many items are found in the current image a "possible overflow" error is produced. If too few items are found a "possible underflow" error is given. As we are unable to discover how many times a loop is executed these must be errors and not warnings.

FLINT may be able to "syntax check" the high level code, stopping the mismatching of control flow words.

7.6 Defining words

Defining words can be broken into two parts. The "Pre-defined" words and the building blocks to allow "user-defined" defining words.

7.6.1 Pre-defined

`CODE` words cannot be checked. Thus the given stack comment is taken to be correct. If no stack comment is given then an error is recorded. The system could be made to give a warning when the `CODE` word was used indicating that the word could not be fully type-checked.

When a method of type checking assembler code is developed, (possibly similar to the type checking system given in chapter 6) it may be possible to provide type checking of `CODE` level words. However, this currently does not exist (to our knowledge), thus we are unable to type check assembler definitions.

`:` will define an entry in the data-base, associating the (checked) stack comment with the word. Later (when the word is used) its stack comment is assumed to be correct. If an error occurred when checking the definition, a warning is given for any word that uses it indicating the uncertain status of the check.

`VARIABLE` defines a word with the stack description of " `--- *w`" (giving a reference to a wildcard). Thus allowing any single-cell type to be stored in the variable. If a stack comment is given then it will be used (provided that it matches with the default).

`2VARIABLE` acts in the same way as `VARIABLE` except that the default stack description is " `--- *dw`" (giving a reference to a double wildcard).

7.6.2 User-defined

`CREATE` will mark the current word as a defining word. Words defined using this word will inherit the run time stack signature associated with the word. The system may also check the syntax definition of the defining word.

`DOES>` will check that the current stack signature matches the stack command given for the word. It will then take the following stack command as the run-time stack signature. The system will check that the typing of the run-time part of the word is correct. Note that the address reference placed on the stack by the `DOES>` word *must* be given in the stack comment.

`;CODE` will check that the current stack signature matches with the signature given for the word. An error is reported if the two signatures do not match. As code entries cannot be checked the run-time stack signature is assumed to be correct. When the defined word is used the system can be made to give a warning indicating the use of an unchecked word.

Thus a defining word may be defined:

```
: CCONST \ syntax: CCONST <char> <word>
         \ stack: ---
   BL WORD 1+ C@ CREATE C,
   DOES>  \ stack: *c --- c
          C@
 ;
```

7.7 Vocabularies

To increase the speed of the system, it will store the checked type signature as part of a word's header definition. As the type signature is being stored along with the word name, the vocabulary structure offered by the host system will automatically be adhered to. Thus, FLINT will need to follow the host systems vocabulary structure. Adoption of the standard vocabulary mechanism (such as the one proposed in the ANSI standard) would be an advantage to this system.

7.8 Error Log

The error log is the output file from the FLINT system. It will consist mainly of error and warning reports.

7.8.1 Error report

The format of the error/warning report will look something like:

```
<level>: <filename> ( <line-no> )
         <word>:-  <message>.
```

where:

<*level*> indicates the level of the error. That is to say that it is "`Error`" or "`Warning`", where "`Error`" is fatal and must be resolved, "`Warning`" is only informative and is probably caused by a previous error.

<*filename*> is the name of the file being processed when the error or warning was discovered. This is the name of the current file, thus if the main file includes sub-files, this will be the name of the sub-file.

< *line-no* > is the line number in the given file upon which the error or warning occurred. More importantly, it is the line at which the error/warning was discovered.

<*word*> is the word definition that was being checked at the time the error or warning was produced.

<*message*> is a single line text message given the error or warning condition.

7.8.2 Verbose reports

If the verbose flag (`-V`) is given on the command line the reports will have four parts:

```
<level>:  <filename> (<line-no>)
          <word> :-  <message>.
          <stack>
          <processed-line>
                          <unprocessed-line>
```

where <*level*>, <*filename*>, <*line-no*>, <*word*> and <*message*> are the same as for the normal error log, <*stack*>, <*processed-line*> and <*unprocessed-line*> are given as:

<*stack*> is the current stack image held by the system at the point of error.

<*processed-line*> shows the error line. The line up to the point at which the error was discovered is shown on this line.

<*unprocessed-line*> shows what is left on the line. If the report was simply a warning, the text on this line will be processed. However, if the report was an error then the rest of this line and indeed the definition is ignored.

It should also be noticed that each error or warning starts at the first character of the line. All subsequent lines are indented, leaving a space at the start of the line and a blank line between errors. The system is designed in this way so as to ease the automatic searching of the error log.

7.8.3 Statistics

At the end of the error log the system will output the following statistics that it has managed to discover during the type checking:

```
<n>  Error[s]
<n>  Warning[s]
```

Note that the optional "`s`" is only left off the word when the number <*n*> is 1. This is correct English and should be provided on all such programs.

7.8.4 Statistics flag

When the `-S` (statistics) command line option is given then the following lines of additional statistical information is also given:

```
<n>  line[s],
<n>  file[s],
<n>  assumption[s],
<n>  assert[s],
<n>  definition[s],
<n>  user defined type[s].
```

Although the information given in these statistics is only concerned with the type checking, there is no reason why the system could not be made to give statistical information, and software metrics (Hand 1988), should the user require it.

7.9 Problems

When this system is implemented there are several design decisions that must be made. They are:

1. How to check if a word's definition will cause a stack underflow to occur and how to report the condition.

2. How to check that a definition will not cause a stack overflow to occur when executed. If it will, how is the condition to be reported.

3. How is the system to handle the error reporting around the `PICK` and `ROLL` words. The system could enforce the use of the `assert:` command before the word and the `assume:` command after the word, reporting an error if they are not presence. Alternatively, it can simply mark the word as being unchecked due to the presents of these words.

4. Whether or not to implement the `syntax:` command. They will also need to add checking to compare the declared syntax is correct compared with the word's definition.

5. The capability to type check "in-to" a structure definition. This will require the definition of a standard structure definition operator within the FORTH system.

As FLINT is mainly concerned with the type checking of FORTH programs, these problems are incidental and could be seen as optional extras.

Chapter 8

The Event Calculus

Formal Specification of Real Time Systems by means of Diagrams and Z schemas

The "Event Calculus" is a diagrammatic notation which provides an easily used means of formally specifying the behaviour of concurrent systems. It can describe synchronous and asynchronous communications, data flow modelling and function application, and the expression of temporal constraints. It has the ability to abbreviate the descriptions of complex state changes, such as data base updates, by means of Z schemas.

The Calculus models system states with sets of parameterised state machines which can communicate via n way synchronisations known as "events". In comparison with process algebras such as CSP, CCS and LOTOS, the calculus is relatively easy for a non specialist to use. It is also easier to present specifications that can be understood by developers who do not have a sophisticated mathematical background. The calculus provides a good control of the level of abstraction used in a model.

In this chapter we introduce the Event Calculus with a series of examples and give a formal mathematical interpretation of the diagram notation.

8.1 Introduction

The Event Calculus resembles process algebras such as CCS (Milner 1989) and CSP (Hoare 1985) in that the mathematical model of communication derives from a simultaneous state change in more than one state machine. We provide a model theoretic description of how the behaviour of a composite state machine can be derived from the behaviour of its component state machines. The basic model is then extended to include asynchronous events, value passing, function application and time. Finally, we introduce the use of supplementary Z schemas to augment the diagram notation and the use of "Schema Transitions" to describe complex state changes such as data base updates.

An important aim of the calculus is that the use of diagrams should give the user a more intuitive and direct view of system behaviour than can be achieved by algebraic expressions alone. Unlike less formal diagram notations, such as DFDs, our diagrams give a complete model of system behaviour and may be thought of as the user interface to an underlying algebra of machine behaviours. Unlike other formal models of concurrency which can also provide a diagrammatic representation of simple processes, the diagram notation for the Event Calculus is fully equipped to deal with complex examples involving parameter passing, function application, asynchronous events and data base updates.

The Event Calculus is particularly well suited for use with the FORTH language. Each FORTH actor (or task) can be represented in the Event Calculus as a state machine.

8.2 State Machines

The Event Calculus is based on state machines which are associated with a set of state changing "events". Our state machines are described in terms of a set of states, a set of events and a next state function which maps a state event pair to a new state.

Our first example consists of a pair of state machines, V and C. The next state functions ψV and ψC are as shown in figure 8.1.

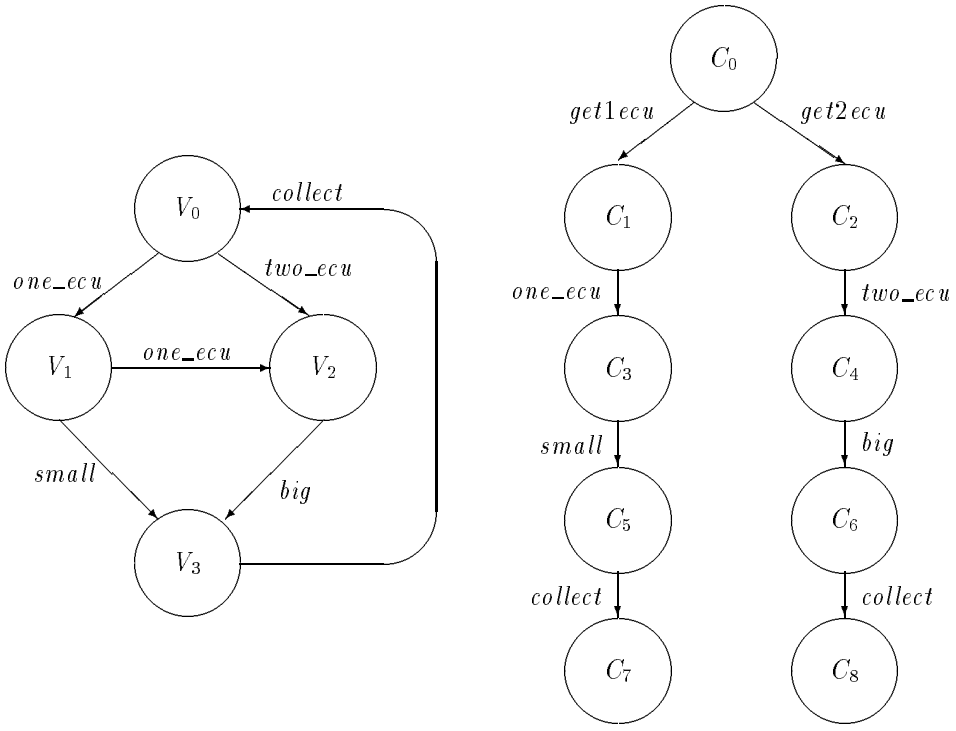


Figure 8.1: The state machines V and C

V is a simple futuristic vending machine which can accept one ecu or two ecu coins and can dispense either a small or big chocolate bar.

$$states\ V = \{V_0, V_1, V_2, V_3\}$$

$$events\ V = \{one\_ecu, two\_ecu, small, big, collect\}$$

C is a child who gets a one ecu coin or a two ecu coin and inserts it into the vending machine to purchase a chocolate bar.

$$states\ C = \{C_0, C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8\}$$

$$events\ C = \{get1ecu, get2ecu, one\_ecu, two\_ecu, small, big, collect\}$$

Some of the events of C, such as one_ecu, are shared with V. When a shared event occurs both C and V will simultaneously move to a new state. We call such an event "synchronous". The child puts the coin in at the same instant as the machine has the coin put in.

We now describe (informally) the Event Calculus rules for determining the behaviour of the composite machine $\{V, C\}$.

Let us suppose we start in state $\{V_0, C_0\}$. In state V_0 machine V is ready to participate in events *one_ecu* and *two_ecu*. However, these are both events that are in the event set of machine C. Events are enabled only when all machines capable of taking part in them are ready to do so. Thus from our initial state these events are disabled and cannot occur.

In state C_0, machine C is ready to perform the events *get1ecu* and *get2ecu*. These events are unique to machine C, so they are enabled (We could also say they can occur because all machines capable of taking part in them are ready to do so, the only such machine being C).

At each stage, any enabled event can occur. Suppose *get1ecu* occurs. The composite machine is now in state $\{V_0, C_1\}$. We denote this state change by writing:

$$\{V_0, C_0\} \xrightarrow{get1ecu} \{V_0, C_1\}$$

Now both C and V are ready to take part in the event *one_ecu*. This is now the only event enabled. In fact the remainder of the behaviour of $\{V, C\}$ is deterministic. All possible composite behaviours of $\{V, C\}$ (there are only two) are shown in figure 8.2.

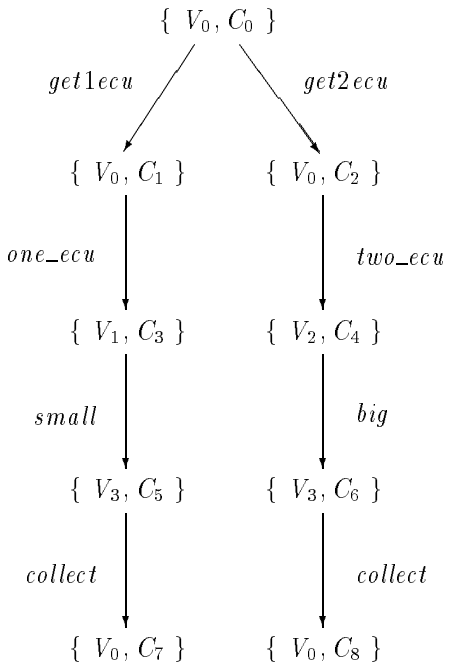


Figure 8.2: Graph of behaviours for the model of figure 8.1

We follow a naming convention that uses machine names derived from the common root of the state names for that machine. Thus the state names V_0, V_1, V_2 and V_3 have a common root V which is the name of the corresponding machine.

8.3 The Formal Model

We now provide formal rules for deriving the behaviour of a set of machines from the next state functions of each individual machine.

Basic types for our discussion are:

$$[MACHINE, STATE, EVENT]$$

$MACHINE$ is the set of state machines (or more precisely state machine identifiers).

$STATE$ is the set of possible machine states.

$EVENT$ is the set of events.

We define a "next state" function:

$$\psi : MACHINE \rightarrow ((STATE \times EVENT) \nrightarrow STATE)$$

The idea here is that ψ is a function from machines to next state functions. Thus ψm will return the next state function for machine m.

We define a function to identify the set of states associated with a particular machine and to specify that each state is associated with only one machine.

$$\begin{array}{|l} states : MACHINE \rightarrow \mathbb{P}\, STATE \\ \hline \forall\, m : MACHINE \bullet \\ \quad states\ m = \mathrm{ran}(\psi m) \cup \mathrm{dom}\,(\mathrm{dom}\,(\psi m)) \\ \forall\, m_1, m_2 : MACHINE \bullet \\ \quad m_1 \neq m_2 \Rightarrow states\ m_1 \cap states\ m_2 = \varnothing \end{array}$$

We define a function to return the unique machine associated with a given state.

$$machine == \{s : STATE;\ m : MACHINE \mid s \in state\ m \bullet s \mapsto m\}$$

The basis of the Event Calculus is the definition of a function χ that describes the behaviour of a set of possibly communicating machines in terms of the next state functions of each individual machine. First, however, we define some functions and sets which we will use to make the definition of χ more readable.

First, we define a function that tells us whether an event can occur when a machine is in a given state.

$$\begin{array}{|l} ready : STATE \times EVENT \rightarrow \mathbb{B} \\ \hline \forall\, e : EVENT;\ s : STATE \bullet \\ \quad ready(s, e) = (s, e) \in \mathrm{dom}\,(\psi\,(machine\ s)) \end{array}$$

Considering our vending machine V as an example:

$$\begin{aligned} ready(V_0, one\_ecu) &= \text{true} \quad \text{as event } one\_ecu \text{ can occur in state } V_0. \\ ready(V_0, big) &= \text{false} \quad \text{as event } big \text{ cannot occur in state } V_0. \end{aligned}$$

The *repertoire* of a machine is the set of all events in which it can participate. The set of *events* associated with the next state function of a machine is a subset of its *repertoire*[1].

$$\begin{array}{|l} events,\ repertoire : MACHINE \rightarrow \mathbb{P}\, EVENT \\ \hline \forall\, m : MACHINE \bullet events\ m = \mathrm{ran}\,(\mathrm{dom}\,(\psi m)) \wedge events \subseteq repertoire \end{array}$$

When considering a set of state machines we think of an event as causing a composite state change which may affect more than one machine. Valid composite states are ones in which each constituent state represents the state of a different machine. Formally, we define the set of valid state sets as follows:

$$\begin{array}{l} validstateset = \\ \quad \{sset : \mathbb{P}\, STATE \mid \forall\, s_1, s_2 : sset \bullet \\ \qquad s_1 \neq s_2 \Rightarrow machine\ s_1 \neq machine\ s_2\} \end{array}$$

[1] This distinction is useful because we may wish to provide a next state function which shows only part of the behaviour of a machine, a subset of *events* from a machines *repertoire*.

In the context of a composite state set, an event may only occur if every machine which has that event in its *repertoire* is ready for it. We define a boolean function that will tell us whether a particular event is enabled for a given composite state.

$$
\begin{array}{|l}
enabled : (validstateset \times EVENT) \to \mathbb{B} \\
\hline
\forall\, sset : validstateset;\ e : EVENT \bullet \\
\quad enabled(sset, e) = \\
\qquad (\forall\, s : sset \bullet e \in repertoire(machine\ s) \Rightarrow ready(s, e)) \\
\qquad \land\ (\exists\, s : sset \bullet ready(s, e))
\end{array}
$$

We now define the function χ which derives the behaviour of a set of state machines from the individual behaviours of each machine plus the enabling rule for composite events. We will define χ to take a set of machines as its argument and to return a next state function for that set of machines. χ is described by giving its domain and by describing its application to an arbitrary element of its domain.

$$
\begin{array}{|l}
\chi : \mathbb{P}\, MACHINE \to ((validstateset \times EVENT) \to validstateset) \\
\hline
\forall\, sset : validstateset;\ e : EVENT;\ mset : \mathbb{P}\, MACHINE \bullet \\
\quad \mathrm{dom}(\chi\, mset) = \{e : EVENT;\ sset : validstateset \mid \\
\qquad machine\ (\!|\ sset\ |\!) = mset \land enabled(sset, e) \bullet (sset, e)\} \\
\quad \land \\
\qquad (sset, e) \in \mathrm{dom}(\chi\, mset) \Rightarrow \\
\qquad\quad (\chi\, mset)(sset, e) = \{s' : STATE \mid \\
\qquad\qquad s' \in sset \land e \notin events(machine\ s')\} \\
\quad \lor \\
\qquad \exists\, s : sset \bullet e \in events(machine\ s) \land s' = (\psi(machine\ s))(s, e)\}
\end{array}
$$

8.4 An Algebra of machine behaviours

The function χ maps from a set of machines to a function which describes the possible composite behaviours of those machines. We now introduce a binary operation to compose such behaviours. We write this operation as $\|$ (pronounced "par").

$$
\begin{array}{|l}
\_\|\_ : range\ \chi \times range\ \chi \to \chi \\
\hline
\forall\, mset_1, mset_2 : \mathbb{P}\, MACHINES \bullet \\
\quad \chi\, mset_1 \| \chi\, mset_2 = \chi(mset_1 \cup mset_2)
\end{array}
$$

χ is now a homomorphism from the algebra of set unions $[\mathbb{P}\, MACHINES, \cup]$ to the algebra of machine behaviours $[range\ \chi, \|]$. It follows that $[range\ \chi, \|]$ is a commutative monoid (ie, $\|$ is commutative, associative and has a unit element).

8.5 Labelled Transitions

For arbitrary $s, t : STATE;\ e : EVENT$ we introduce the notation:

$$s \xrightarrow{e} t$$

to indicate that the machine associated with state s goes from state s to state t when event e occurs. We call this a "labelled transition". Formally the notation is introduced as follows.

$$
\begin{array}{|l}
\_ \overset{\_}{\Leftrightarrow} \_ : STATE \times EVENT \times STATE \to \mathbb{B} \\
\hline
s \overset{e}{\Leftrightarrow} t \Leftrightarrow (\psi(machine\ s))(s, e) = t
\end{array}
$$

8.6 Simple Examples

In this section we provide some simple examples of Event Calculus models and at the same time introduce some additional notations for use in Event Calculus diagrams.

8.6.1 The specification of mutual exclusion (without fairness)

Consider two processes A and B, and a semaphore S.

$$states\ A = \{A_0, A_1\}$$
$$states\ B = \{B_0, B_1\}$$
$$states\ S = \{S_0, S_1\}$$

A_1 and B_1 are the critical regions of processes A and B.

S will be in state S_1 when one of the processes is inside its critical region and in state S_0 otherwise.

Starting from initial composite state $\{A_0, B_0, S_0\}$, any composite state which includes $\{A_1, B_1\}$ will be impossible to reach.

The individual machines are as shown in figure 8.3.

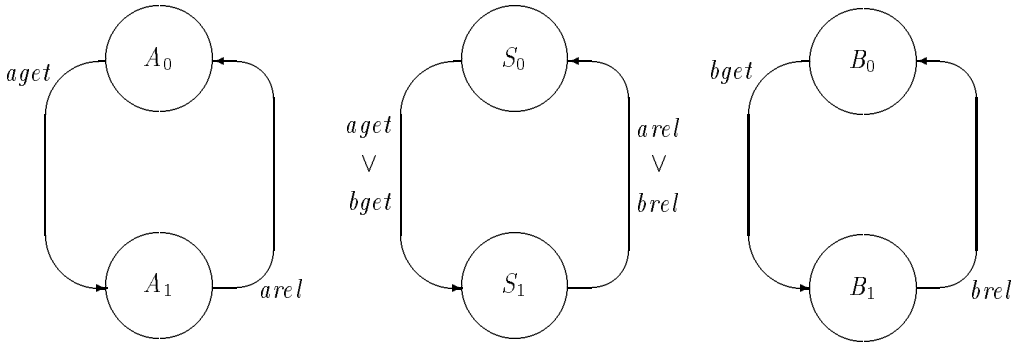


Figure 8.3: Mutual exclusion without fairness

This diagram introduces a convention by which two alternative transitions with the same start and end states are shown by a single line with an appropriate label. For example the arc from S_0 to S_1 which is labelled $aget \vee bget$.

Suppose we start the model in composite state $\{A_0, B_0, S_0\}$. The possible transitions for the composite state are as shown in figure 8.4.

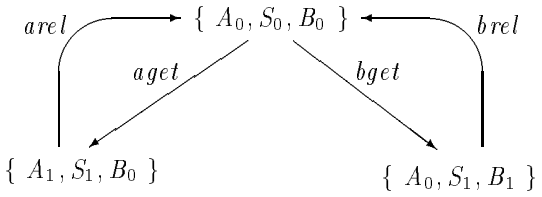


Figure 8.4: Graph of behaviours for the model given in figure 8.3

When two (or more) alternative transitions are available (in this case *aget* and *bget*) it is upto the designer to fire and trace the events as appropriate (see the trace on page 72 of section 8.6.4 for an example of tracing events).

If we were to give machine A precedence over machine B, we would observe the unfair nature of this model, in that machine B will *never* be capable of reaching state B_1. The event *bget* is only valid when we have the composite state $\{A_0, B_0, S_0\}$, however the event *aget* is also valid in this state. As we are favouring machine A over machine B, we now fire the *aget* event, thus preventing the *bget* event from ever being fired (a condition referred to as *indefinite postponement*).

8.6.2 Asynchronous Events

Machine A broadcasts job requests to machines B and C. The broadcast event, which we denote by *job*, is to be asynchronous. That is, B and C do not have to be ready for the broadcast for it to occur. We distinguish such asynchronous events in our diagram notation by underlining them in the graph of the state machine which originates them.

In this simple model, A knows that it can have two jobs outstanding at any one time and will not attempt to broadcast a further job until one of these is done (see figure 8.5).

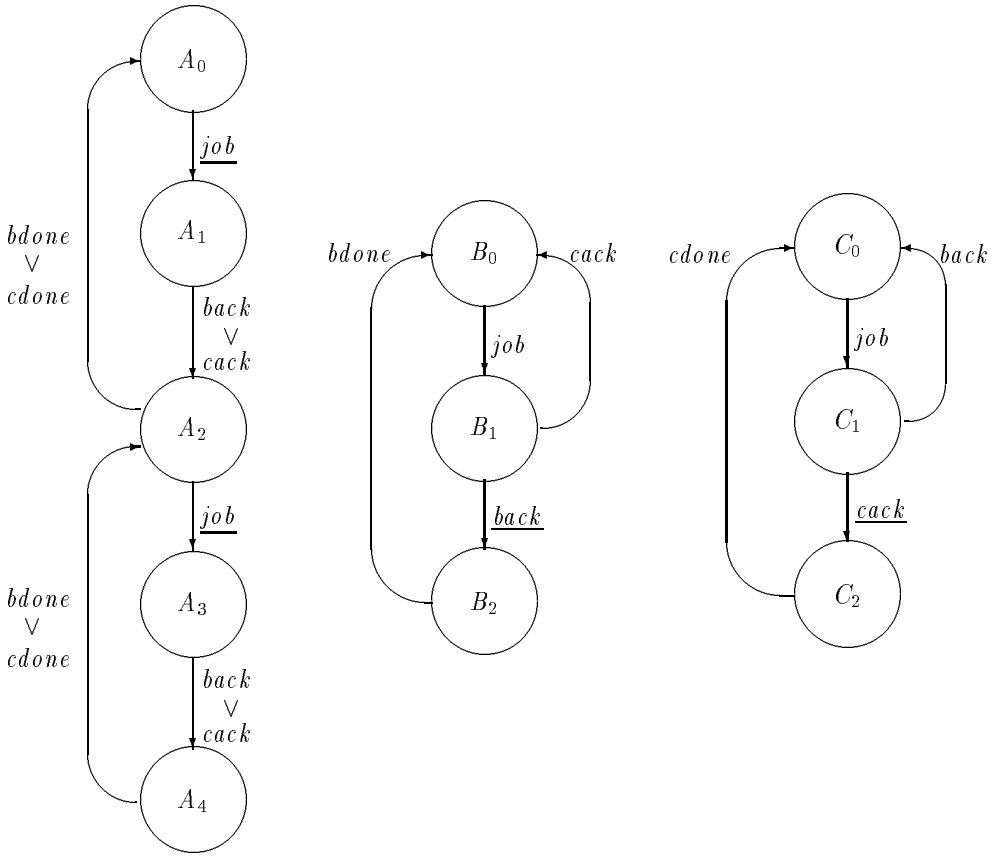


Figure 8.5: An example using asynchronous broadcasts

Now consider the following composite state transitions from an initial composite state $\{A_0, B_0, C_0\}$

$$\{A_0, B_0, C_0\} \xrightarrow{job} \{A_1, B_1, C_1\} \tag{8.1}$$

$$\{A_1, B_1, C_1\} \xrightarrow{back} \{A_2, B_2, C_0\} \tag{8.2}$$

$$\{A_2, B_2, C_0\} \xrightarrow{job} \{A_3, B_2, C_1\} \tag{8.3}$$

$$\{A_3, B_2, C_1\} \xrightarrow{cack} \{A_4, B_2, C_2\} \tag{8.4}$$

According to our laws for composite transitions, the composite transitions 8.3 and 8.4 cannot occur, as they are not synchronised. To allow asynchronous events to be declared we must add some additional structure to our calculus.

One possibility would be to distinguish two types of event (synchronous and asynchronous) and formulate rules for the underlying calculus accordingly. Another approach, which we follow here, is to describe asynchronous events in terms of the existing calculus. In terms of the underlying formalism, the effect of declaring *job* as an asynchronous event originated by machine A is to add some additional null transitions to B and C to allow *job* to occur in all states of B and C.

Let the original next state function as given by the above graphs be ψ_0. We construct ψ_1 which allows *job* to be asynchronously originated by A:

$\psi_1\, m =$
  if $m = A$ then $\psi_0\, A$
  else $\bigcup_{s \in states\ m} \{(s, job) \mapsto s\} \oplus \psi_0$

The idea is to add labelled transitions of the form $s \xrightarrow{job} s$ to the next state functions of B and C for all states s which are not otherwise ready to participate in the event *job*.

The event *back* is asynchronously originated by B and event *cack* is asynchronously originated by C. We can construct appropriate next state functions to express this as follows.

From ψ_1 we can construct ψ_2 which allows *back* to be asynchronously originated by B and from ψ_2 construct ψ_3 which allows *cack* to be asynchronously originated by C.

In general, we describe a constructor function *async* which takes a next state function ψ, a machine m^*, an event e and returns a new next state function $async(\psi, m^*, e)$ which has the additional labelled transitions required to allow e to be asynchronously originated by m^*.

$SIMPLE\_NEXT\_STATE\_FUNCTION ==$
  $MACHINE \rightarrow (STATE \times EVENT) \nrightarrow STATE$

$async : SIMPLE\_NEXT\_STATE\_FUNCTION \times MACHINE \times EVENT$
  $\nrightarrow SIMPLE\_NEXT\_STATE\_FUNCTION$

$\forall\, \psi : SIMPLE\_NEXT\_STATE\_FUNCTION;$
$m, m^* : MACHINE;$
$e : events\ m^* \bullet$
  $e \in events\ m \land$
  $async(\psi, m^*, e)\, m =$
    if $m = m^*$ then $\psi\, m$
    else $\bigcup_{s : states\ m} \{(s, e) \mapsto s\} \oplus \psi\, m$
  $\lor$
  $e \notin events\ m \land$
    $async(\psi, m^*, e)\, m = \psi\, m$

8.6.3 Value passing

Our calculus is based on a primitive notion of synchronised events which do not, of themselves, admit any notion of direction in communication. However, we can build such a notion and use it to express ideas such as value passing and function application. The basic idea is taken from the value passing calculus of Ccs.

Consider the state machines shown in figure 8.6, where A may start in initial state $A0_0$ or $A0_1$, and B starts in initial state $B0$.

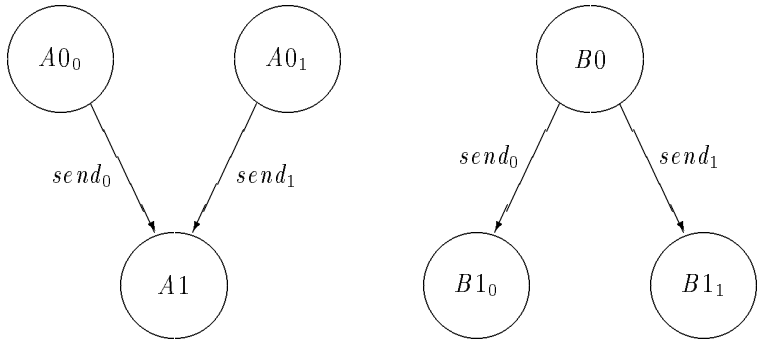


Figure 8.6: Primitive value passing

Depending on the initial state of A, the possible events are $send_0$ or $send_1$ which lead to a final state for B of $B1_0$ or $B1_1$ respectively.

We can think of events $send_0$ and $send_1$ as conveying state information from machine A to machine B. This idea is the basis of the value passing calculus.

Given the following schema text:

$$
\begin{array}{|l}
X : \mathbb{P}\,\mathbb{N} \\
x : X \\
A0, B1 : X \rightarrowtail STATE \\
send : X \rightarrowtail EVENT \\
\hline
X = \{0, 1\} \\
A0 = \{0 \mapsto A0_0, 1 \mapsto A0_1\} \\
B1 = \{0 \mapsto B1_0, 1 \mapsto B1_1\} \\
send = \{0 \mapsto send_0, 1 \mapsto send_1\}
\end{array}
$$

We will be able to show the simple example given in figure 8.6 as in figure 8.7. In the above schema, $A0$ and $B1$ are declared as (partial) injections because each element in their domain must map to a different state. Similarly $send$ is declared as a (partial) injection because each element in its domain must map to a different event. Such partial injections identify, by their range, a family of states or events, and we will refer to, for example, "the family of states $A0$" or "the family of states $A0(x)$" (where x is an arbitrary member of the domain of $A0$). In future, we will not name individual states or events from such families but will refer to them by means of the functions that map onto them (eg, we refer to $A0(0)$ rather than to an actual state name such as $A0_0$).

8.6.4 Mutual Exclusion with fairness

Processes A and B submit requests to enter their critical regions by adding an identity token to a two place queue modelled by QA and QB. The requests are granted when the tokens are removed from

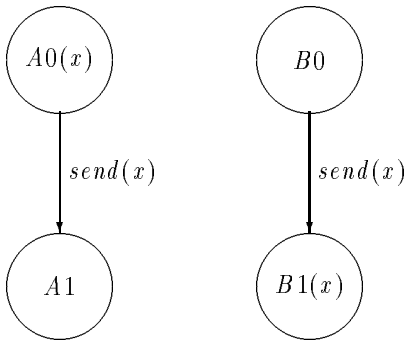


Figure 8.7: Parameterised value passing

the other end of the queue. Entry to the critical region is governed by a semaphore S. When a process is in its critical region the state of S records this and also records the identity of the process.

The event calculus diagram for the model is given in figure 8.8. Some event identifiers ($req(p)$) represent a family of events and some state identifiers ($QA(p)$) represent a family of states. Since the domains of these injections contain two members, these families each contain two members as well. The diagram includes basic declarations of the functions req, QA etc. In an actual formal specification these declarations would be in schema form with additional predicates defining, for example, the exact domains of such functions. In this chapter such details have been omitted.

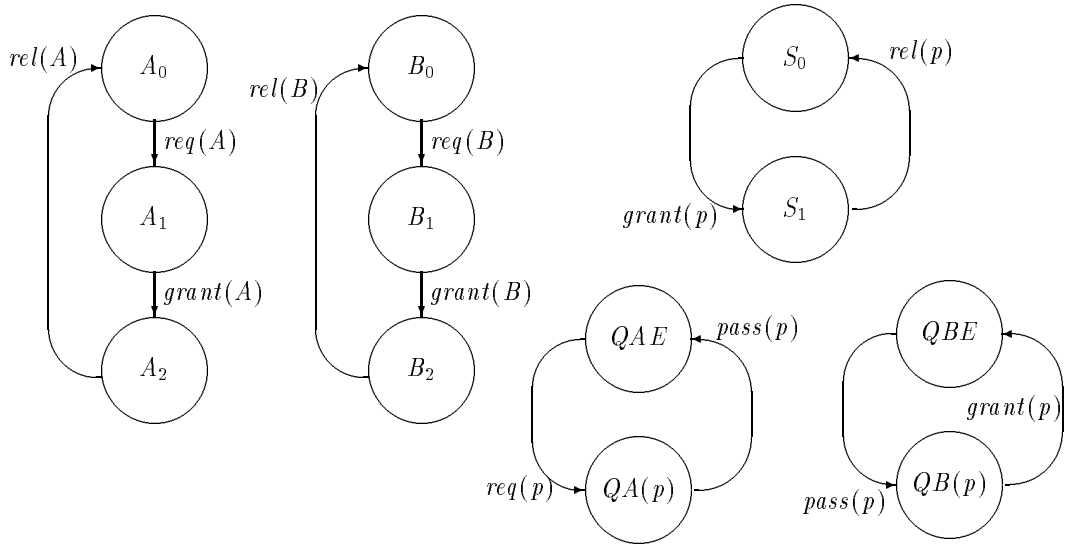
We could write out a trace of events from a the model as follows:

Event	State
	$\{A_0, B_0, S_0, QAE, QBE\}$
$req(A)$	$\{A_1, B_0, S_0, QA(A), QBE\}$
$pass(A)$	$\{A_1, B_0, S_0, QAE, QB(A)\}$
$req(B)$	$\{A_1, B_1, S_0, QA(B), QB(A)\}$
$grant(A)$	$\{A_2, B_1, S_1(A), QA(B), QBE\}$
$pass(B)$	$\{A_2, B_1, S_1(A), QAE, QB(B)\}$
$rel(A)$	$\{A_0, B_1, S_0, QAE, QB(B)\}$
$req(A)$	$\{A_1, B_1, S_0, QA(A), QB(B)\}$
$grant(B)$	$\{A_1, B_2, S_1(B), QA(A), QBE\}$

This model is considered "fair" where the model show in figure 8.3 is "without fairness". If we were to take the same disposition as we do in section 8.6.1, ie, if we were to prefer machine A over machine B, we would have different results. When machine A enters into state A_2, we are in a position of being able to fire the $req(B)$ event (as shown in the above trace). Thus machine B is now able to enter into state B_2 (after machine A releases the semaphore with a $rel(A)$ event, as shown in the above trace). Machine B is now sure to get a "turn" in its critical region before machine A gets its next trun.

8.7 A GCD algorithm, modelling parameter passing and procedure call

The greatest common divisor function may be defined recursively as follows:



Declarations:
$S_1, QA, QB : MACHINE \rightarrowtail STATE$
$req, grant, rel, pass : MACHINE \rightarrowtail EVENT$

Figure 8.8: Mutual exclusion with fairness

$$gcd : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

$$\forall\, x, y : \mathbb{N} \bullet$$
$$gcd(x, 0) = x$$
$$gcd(x, y) = gcd(y, x \bmod y)$$

Figure 8.9 provides a composite state machine for calculating a greatest common divisor. Machine A models a "main program" and machine B models a procedure for calculating $x \bmod y$.

To see how function application is being modelled, consider the family of transitions:

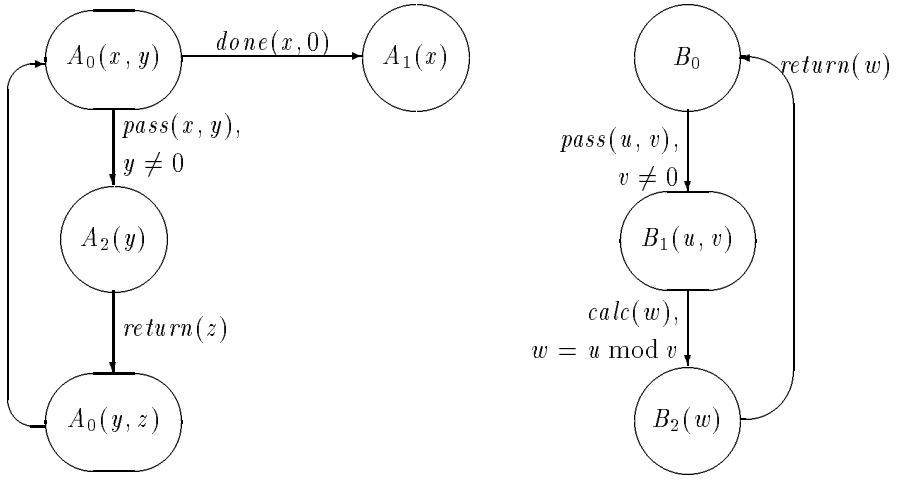
$$B_1(u, v) \xrightarrow{calc(w)} B_2(w) \quad (w = u \bmod v)$$

Each different (u, v) pair is associated with a different state in the family of states $B_1(u, v)$. With each of these states we associate an event $calc(w)$ where $w = u \bmod v$. For each w the occurrence of event $calc(w)$ will take machine B into state $B_2(w)$.

In drawing machine A we have included an unnamed transition from $A_0(y, z)$ to $A_0(x, y)$. We think of this as an instantaneous unsynchronised transition in which nothing changes except the parameter names. The old value of y becomes the new value of x, and the old value of z becomes the new value of y.

8.8 Variables and Scopes

Consider machine A as given in figure 8.9. We appear to be able to follow the state history of variables x and y as the "program" progresses. Although this intuition is correct, is should be underpinned with an appreciation of the underlying formalism. Formally the diagram for machine A declares the following families of labelled transitions.



Declarations:

$A_0, B_1 : \mathbb{N} \times \mathbb{N} \rightarrowtail STATE$
$A_1, A_2, B_2 : \mathbb{N} \rightarrowtail STATE$
$done,\ pass : \mathbb{N} \times \mathbb{N} \rightarrowtail EVENT$
$calc,\ return : \mathbb{N} \rightarrowtail EVENT$

Figure 8.9: GCD algorithm, with subroutine call

$$A_0(x, y) \xrightarrow{done(x, 0)} A_1(x) \tag{8.5}$$

$$A_0(x, y) \xrightarrow{pass(x, y)} A_2(y) \tag{8.6}$$

$$A_2(y) \xrightarrow{return(z)} A_0(y, z) \tag{8.7}$$

Consider transitions 8.6 and 8.7 of these. The identifiers x and y used in 8.6 are bound variables local to this family of transitions. The y in 8.7 is a separate bound variable associated with a different expression. As with all bound variables, the choice of identifier names is arbitrary and formally we could replace the y in 8.7 with any identifier name except x (which is already spoken for within the same scope).

However, the diagram notation we use limits our arbitrary choice of identifier name in a way that supports the intuitive understanding we have of persisting identifier values. According to this understanding, the y in 8.6 can be thought of as the same y as in 8.7 since in any trace they will take the same value when an event from 8.6 is followed by an event from 8.7.

8.9 Time

We measure time in the Event Calculus in terms of clock ticks. A clock tick is a kind of universal cosmic heartbeat, which differs from an event both in its universality and in being free of any synchronisation requirements. States may be associated with a clock function, which records the number of ticks that have occurred since that state came into existence.

We will place timing constraints on states by means of two partial functions which give the minimum and maximum times required for given events to occur after entering a given state. These will be partial functions because not all states and events will have timing constraints.

We also introduce the set *timed* which is the set of all time constrained $STATE \times EVENT$ pairs and the set *timed_states*, which is the set of states which have any time constrained events.

$$
\begin{array}{|l}
minreq, maxreq : STATE \times EVENT \rightarrow \mathbb{N} \\
timed : STATE \times EVENT \\
timed\_states : \mathbb{P}\, STATE \\
\hline
timed = \mathrm{dom}\ minreq = \mathrm{dom}\ maxreq \\
\quad \wedge \\
timed\_states = \{s : STATE \mid \exists\, e : EVENT \bullet (s, e) \in timed\} \\
\quad \wedge \\
\forall\, s : STATE;\ e : EVENT \bullet ((s, e) \in timed \Rightarrow minreq\ s \leq maxreq\ s) \\
\quad \wedge \\
\forall\, s : STATE;\ e : EVENT \bullet (s, e) \in timed \Rightarrow \neg\, (s \overset{e}{\nrightarrow} s)
\end{array}
$$

In the next section we will formally introduce a clock function as part of the system state. States which are subject to timing constraint have an associated clock which records how long they have been in existence, any new time constrained state that results from the event has its clock initialised to zero. Note that the final predicate of the above schema is to disallow null transitions from being time constrained. This is to avoid having to decide whether a null transition should reset a state clock.

Since timing constraints place additional restrictions on whether events can occur, we will henceforth distinguish "enabled events" (those capable of occurring if timing restraints are not considered) from "firable events" (those capable of occurring given that timing constraints are to be taken into consideration). All firable events are necessarily enabled, but not all enabled events need be firable.

In defining what we mean by a firable event we make use of the functions *minreq* and *maxreq* as follows. A time constrained state s must exist for at least $minreq(s, e)$ ticks before event e can occur. After s has persisted for $minreq(s, e)$ ticks the event e may occur if it is firable, but another clock tick (or another enabled event) may occur instead. After s has persisted for $maxreq(s, e)$ ticks or longer, the event e will occur before the next clock tick if it can. However, another tick may occur if e is not enabled. Even if e is enabled it may not occur as there may be other firable events which may occur instead.

Figure 8.10 shows a simple mutual exclusion model with time constraints on the states of machines A and B.

Figure 8.11 shows a partial graph of the possible behaviours of the composite machine shown in figure 8.10. In this graph we use the notation (s, n) to show that the timed state s has persisted for n clock ticks. Thus $(A_0, 2)$ indicates that state A_0 has persisted for 2 clock ticks.

Initially, although the events $get(A)$ and $get(B)$ are enabled, they are not firable because of timing constraints. After the first clock tick the event $get(B)$ becomes firable. We now have a choice of possible behaviours consisting of another clock tick or the event $get(B)$.

The trace consisting of "*tick*, *tick*", brings us to a composite state where B_0's clock has reached $maxreq(B_0, get(B))$. Some event must now occur before the next clock tick. Two events are firable, $get(A)$ and $get(B)$. If $get(B)$ now occurs we enter the composite state:

$$\{(A_0, 2), S_1(B), (B_1, 0)\}$$

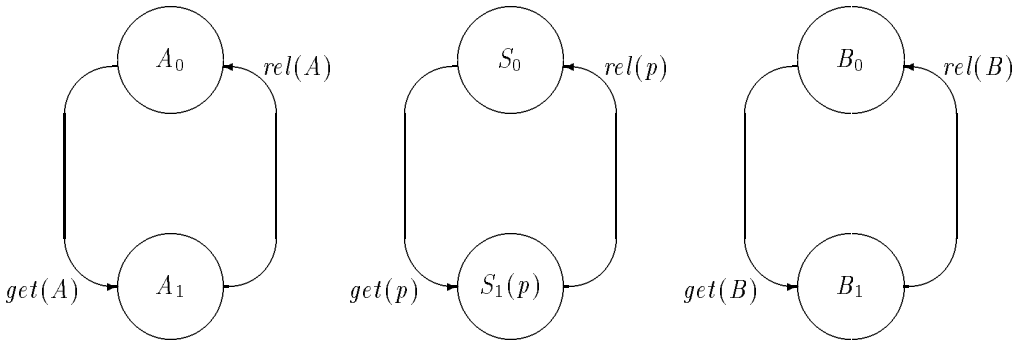
State B_1 is a new member of the compound state and has its clock initialised to zero.

If, on the other hand, $get(A)$ occurs we enter the composite state:

$$\{(A_1, 0), S_1(A), (B_0, 2)\}$$

Now although B_0's clock has reached $maxreq(B_0, get(B))$ the event $get(B)$ is not enabled. At this point a tick can occur, or the event $rel(A)$ can occur.

The form of the timing constraints was chosen to allow the modelling of interrupts, time outs and to allow maximum and minimum times for complex behaviours to be calculated. We base these



Declarations:

$get,\ rel : MACHINE \rightarrowtail EVENT$

$S1 : MACHINE \rightarrowtail STATE$

Timing Constraints:

$minreq = \{((A_0, get(A)), 2), ((A_1, rel(A)), 0), ((B_0, get(B)), 1), ((B_1, rel(B)), 1)\}$

$maxreq = \{((A_0, get(A)), 3), ((A_1, rel(A)), 1), ((B_0, get(B)), 2), ((B_1, rel(B)), 1)\}$

Figure 8.10: Timing constraints example

calculations on the maximum and minimum times required for the possible event sequences that make up the behaviour.

Consider the event sequence:

$$get(A),\ rel(A),\ get(B),\ rel(B)$$

We can deduce the minimum time this sequence can take by generating a trace which includes these events and which will always prefer an event to a clock tick:

$$tick,\ tick,\ get(A),\ rel(A),\ get(B),\ tick,\ rel(B)$$

and we can deduce the maximum time by generating a trace which always prefers a tick to an event:

$$tick,\ tick,\ get(A),\ tick,\ rel(A),\ get(B),\ tick,\ rel(B)$$

A formal specification of the timing rules for the system requires a notion of current system state, which is given in the following section.

8.10 The Dynamic model

Until now our formalisation of the Event Calculus has used a static model, in the sense that our next state functions have held information from which all possible behaviours of an Event Calculus model could be deduced.

We now supplement this static description with a dynamic model which has a notion of the current system state. As part of this model we introduce schemas to describe how the current state is updated by the occurrence of an event or a clock tick.

The "data base schema" for the dynamic model records the composite current state and the clock value of each time constrained state within this composite state.

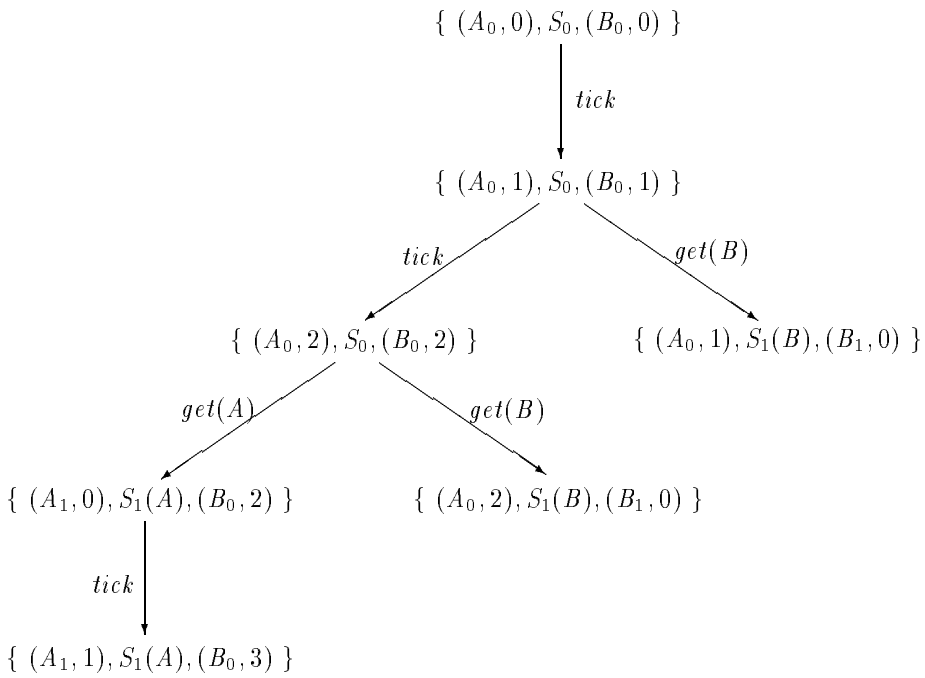


Figure 8.11: Partial graph of behaviours for the machine of figure 8.10

MachineState

$$
\begin{array}{l}
machines : \mathbb{P}\, MACHINE \\
current\_state : validstateset \\
clock : (timed\_states \cap current\_state) \pfun \mathbb{N} \\
firable : EVENT \fun \mathbb{B} \\
\hline
machine\ (\!|\ current\_state\ |\!) = machines \\
\forall\, e : EVENT \bullet \\
\quad firable(e) = enabled(current\_state, e) \\
\quad \land \\
\quad \forall\, s : current\_state \bullet \\
\qquad (s, e) \in timed \Rightarrow clock\ s \geq minreq(s, e)
\end{array}
$$

In this schema we declare the set of machines to be modelled, a set of states which gives the current state of each of these machines and a clock function to give the clock value associated with each time constrained state. We also declare a boolean function which tells us whether a given event may occur (is firable) in the given system state. An event may fire if it is enabled and if all time constrained states that will change as a result of the event have clocks that are past their minimum tick values.

We next describe the changes caused by firing an event.

FireEvent

$\Delta MachineState$
$e? : EVENT$
$result! : \{\text{“}\texttt{ok}\text{”}, \text{“}\texttt{cannot fire}\text{”}\}$

$firable\ e?$
$\quad \land\ current\_state' = (\chi\, machines)(current\_state, e?)$
$\quad \land\ clock' = ((current\_state) \lhd clock) \cup$
$\quad\quad \{s : timed\_states \mid s \in current\_state' \setminus current\_state \bullet s \mapsto 0\}$
$\quad \land\ result! = \text{“}\texttt{ok}\text{”}$
$\lor$
$\quad \neg\ firable\ e?$
$\quad \land\ current\_state' = current\_state$
$\quad \land\ clock' = clock$
$\quad \land\ result! = \text{“}\texttt{cannot fire}\text{”}$

Tick

$\Delta MachineState$
$result! : \{\text{“}\texttt{ok}\text{”}, \text{“}\texttt{an event must occur before the next tick}\text{”}\}$

$current\_state' = current\_state$
$\quad \land\ (\neg\ \exists\, e : EVENT \bullet (firable\ e \land$
$\quad\quad \exists\, s : \mathrm{dom}\ clock \bullet clock\ s \geq maxreq(s, e)) \land$
$\quad clock' = \{s : STATE, n : \mathbb{N} \mid s \mapsto n \in clock \bullet s \mapsto n + 1\}$
$\quad \land\ result! = \text{“}\texttt{ok}\text{”})$
$\lor$
$\quad (\exists\, e : EVENT \bullet firable\ e \land$
$\quad\quad \exists\, s : \mathrm{dom}\ clock \bullet clock\ s \geq maxreq(s, e)$
$\quad \land\ clock' = clock$
$\quad \land\ result! = \text{“}\texttt{an event must occur before the next tick}\text{”})$

8.11 Combining the Event Calculus with Z schema calculus

Data base operations are typically described in Z by first giving a data base schema:

$$S_0 \mathrel{\widehat{=}} [\, D_0 \mid P_0 \,]$$

then defining operations on this data base with schemas having a general form:

$$S_1 \mathrel{\widehat{=}} [\, \Delta S_0;\ D_1 \mid P_1 \,]$$

The declaration D_0 will give the data structures of the data base. The predicate P_0 will describe data base invariants. D_1 will declare Input/Output identifiers associated with the update. P_1 will give any restrictions and preconditions on the form of these identifiers and will also describe how the new system state D_0' is related to the previous system state D_0.

Suppose that in an event calculus model the state of this data base is maintained by a machine M. We will provide an interpretation for a "Schema Transition" of the form:

$$M_0(S_0) \xrightarrow{S_1} M_1(S_0')$$

such that the Schema Transition describes, in terms of the event calculus, the data base update described by the schema S_1 in Z.

In formulating this interpretation we face certain difficulties. Firstly, in the Event Calculus, parameterised events accept tuples as arguments and identify arguments by their position within the tuple. In Z schemas, on the other hand, declarations do not have any particular order and arguments are identified by name. To handle this problem, we extend Z with an alphabetic ordering symbol α so that if D is a declaration, αD will be the same declaration with its components written in alphabetic order by identifier name. Thus if X and Y are basic types and:

$$D \text{ is } x, z : X;\ y : Y$$

then

$$\alpha D \text{ is } x : X;\ y : Y;\ z : X$$

In addition, we use ΘD to represent the type obtained from the declaration D by taking the cartesian product of the types of its identifiers in the order in which they are written, following Spivey (1989) we use θD to represent the characteristic tuple formed by writing out the identifiers of D. For example

$$\Theta(\alpha D) \text{ is } X \times Y \times X$$

and

$$\theta(\alpha D) \text{ is } (x, y, z)$$

These notations will allow us to construct certain tuples required in our event calculus model.

There is also a problem with respect to identifier scope. In a parameterised labelled transition, identifiers are bound variables the scope of which is the labelled transition together with any qualifying predicate. Thus

$$A_0(x) \xrightarrow{step(y)} A_1(z), \quad (y < x \wedge z = x - y)$$

could equally well be written as:

$$A_0(a) \xrightarrow{step(b)} A_1(c), \quad (b < a \wedge c = a - b)$$

In a schema, on the other hand, any identifiers declared in the schema have a scope which lasts till the end of the schema but may subsequently be reintroduced into the formal discussion by quoting the schema name. To overcome this discrepancy we use universal schema quantification. If D is a declaration restricted by a property P and if S is a schema, the declaration part of which includes all the identifiers of D such that S can be expressed in the form:

$$S \mathrel{\widehat{=}} [\, D;\ D_s \mid P_s \,]$$

then

$$\forall D \mid P \bullet S$$

represents the schema

$$[\, D_s \mid (\forall D \mid P \bullet P_s) \,]$$

In our usage of this notation, D and P will be the declarations and predicate of the schema describing the data base update. D_s and P_s are the additional declarations and predicates required to describe the labelled transition which will perform the data base update in the event calculus model.

We need some final conventions to generate the declarations of the parameterised states and the parameterised event needed in the event calculus model. We derive their names from those used in the schema transition

$$M_0(S_0) \xrightarrow{S_1} M_1(S_0')$$

We take M_0 and M_1 as the names of our parameterised state functions. We obtain the parameterised event function name by converting the first character of the data base update schema S_1 to lower case. We represent this name informally as s_1. This usage is informal because it does not show the derivation of the name s_1 from the name S_1. For example, if the name of the data base update schema is *Book*, the corresponding parameterised event name is *book*.

We are now ready to give the event calculus interpretation of the schema transition:

$$M_0(S_0) \xrightarrow{S_1} M_1(S_0')$$

where S_0 is a data base description schema

$$S_0 \mathrel{\widehat=} [\, D_0 \mid P_0 \,]$$

and S_1, which describes a data base update operation, has the form:

$$S_1 \mathrel{\widehat=} [\, \Delta S_0, D_1 \mid P_1 \,]$$

Our interpretation of this text at the event calculus level will be:

$$\forall\, D_0;\ D_0';\ D_1 \bullet S$$

where

S

S_1;
$M : MACHINE$
$M_0, M_1 : \Theta\alpha\, D_0 \nrightarrow states\ M$
$s_1 : \Theta\alpha\, D_1 \nrightarrow EVENT$

$$M_0(\theta\alpha\, D_0) \xrightarrow{s_1(\theta\alpha\, D_1)} M_1(\theta\alpha\, D_0')$$

8.12 A Distributed seat booking system

As an example of the techniques described above, we specify a simple seat booking system in which bookings can be made from a number of different nodes. We use Z schemas to describe the data base and the update and enquiry operations to be performed upon it. We use an Event Calculus diagram to describe the manner in which each node is able to gain access to the data base. Together with the rules of interpretation given above, these two parts of the system description generate a formal model, in Z, of the next state functions for the system's component state machines.

We introduce two basic types, the set of all seats and the set of names for people who may book the seats.

$[SEAT, NAME]$

The state of the data base is described in the following schema using a function from seats to names. Seats which are not in the domain of this function are still free. There is an implicit invariant which disallows double booking of a seat. This invariant arises from the declaration of the relationship between seats and names as a partial function.

Booking

$$booked : SEAT \pfun NAME$$
$$free : \mathbb{P}\, SEAT$$

$$free = SEAT \setminus \mathrm{dom}\ booked$$

We specify an enquiry operation which will return the set of free seats.

Enquire

$$\Xi Booking$$
$$free! : \mathbb{P}\, SEAT$$

$$free! = free$$

We specify an operation to book a seat. The operation takes as inputs a seat and a name. It has the precondition that the given seat should not be already booked and, if this is satisfied, it adds the new booking to the data base. There will be no need to specify what happens if the seat is already booked, as the event calculus part of the specification will render this event impossible by specifying a data validation check.

Book

$$\Delta Booking$$
$$seat? : SEAT$$
$$name? : NAME$$

$$seat? \in free$$
$$booked' = booked \cup \{seat? \mapsto name?\}$$

We now associate these schema with an Event Calculus diagram that models a system in which multiple nodes are able to share access to the data base. Figure 8.12 shows an Event Calculus diagram which, together with the seat booking specification, provides the formal specification for the system.

The seat bookings data base is maintained by machine A. There are two possible operations which can be performed on the data base, a seat booking and an enquiry. These are denoted in the Event Calculus diagram by writing:

$$Book \lor Enquire$$

This declares the two schema transitions:

$$A_0(Booking) \xrightarrow{Book} A_0(Booking')$$

and

$$A_0(Booking) \xrightarrow{Enquire} A_0(Booking')$$

There is a possible ambiguity of notation here since the expression

$$Book \lor Enquire$$

could also be taken to be a single schema formed by the "schema or" of *Book* and *Enquire*. To avoid this, we do not allow schema expressions other than schema names to be written on an event calculus diagram.

The data base update that occurs when a seat is booked is described by the schema transition:

$$A_0(Booking) \xrightarrow{Book} A_0(Booking')$$

The meaning we give to this schema transition can be expressed as:

$$\forall D \bullet S$$

Where D is the declaration part of the *Book* schema and S is the schema:

S

$Book$

$A : MACHINE$

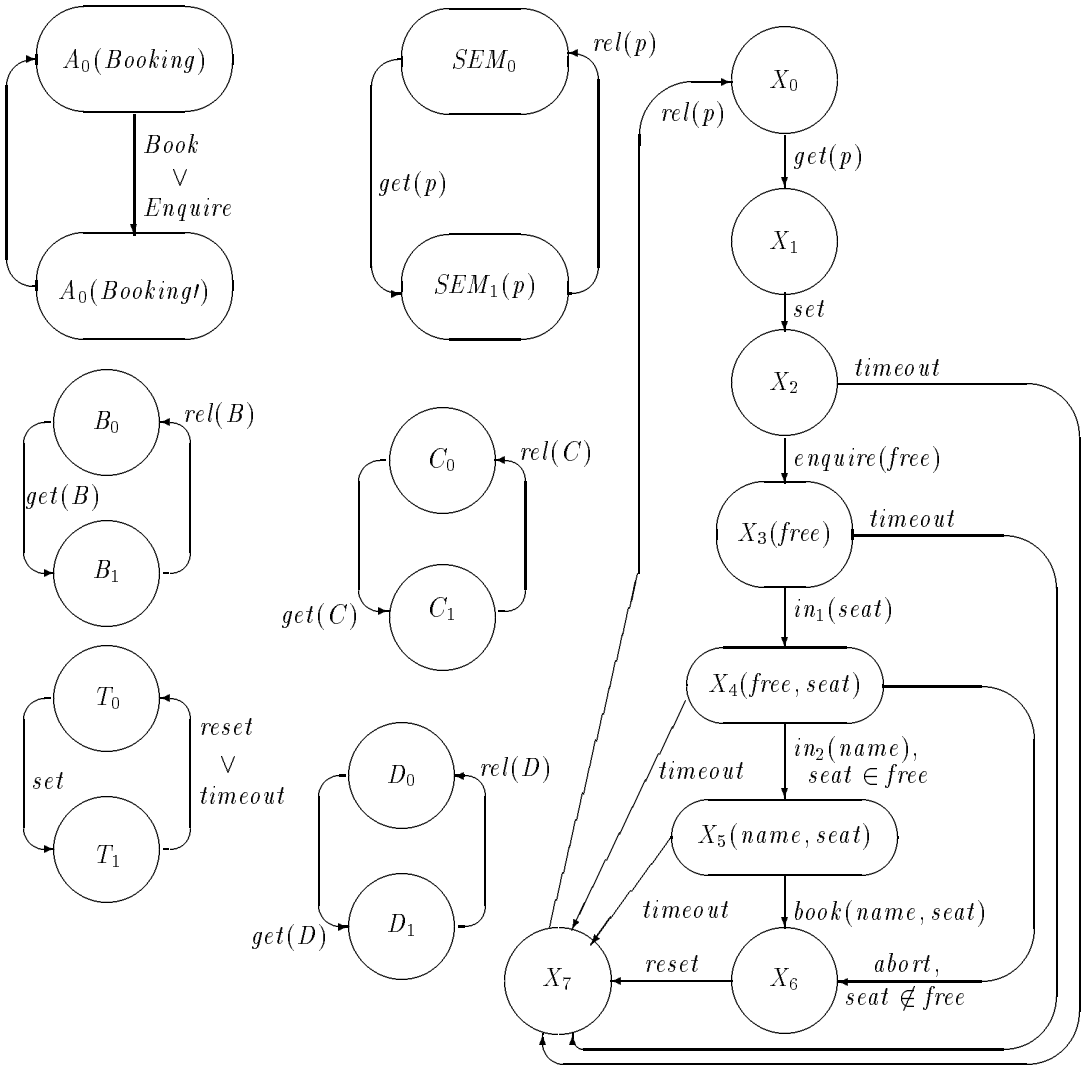
$A_0 : (SEAT \pfun NAME) \times \mathbb{P} SEAT \pfun states\ A$

$book : NAME \times SEAT \pfun EVENT$

$$A_0(booked, free) \xrightarrow{book(name?,\ seat?)} A_0(booked', free')$$

Machines B, C and D are three nodes which originate booking transactions. B performs its bookings when in state B_1, C in state C_1 and so on. Since access to states B_1, C_1 and D_1 is mutually exclusive, the details of the booking operation are shown in a single machine X which runs whenever B is in state B_1 or C is in state C_1 or D is in state D_1. Our intention is for X to be thought of as the common logic shared by all booking nodes, not as a separate machine invoked by a booking node when it wishes to perform a transaction.

SEM is provided to control the mutually exclusive access to machine X, while T functions as a watchdog timer which will cause the booking operation to be forcibly aborted if it cannot be completed in a set time. This is achieved by placing a time constraint on the *timeout* event associated with state T_1 and on the *timeout* event associated with various states of machine X. In our specification, it is actually possible for machine X to continue with its seat booking operations when T is ready to *timeout*. However, the booking operations are not allowed to take any observable time (they must all be completed before the next clock tick).



Declarations:	Time Constraints:
$A_0 : (SEAT \pfun NAME) \times \mathbb{P}\, SEAT \rightarrowtail states\, A$	
$X_4 : \mathbb{P}\, SEAT \times SEAT \rightarrowtail states\, X$	
$X_5 : NAME \times SEAT \rightarrowtail states\, X$	$minreq(T_1, timeout) = 10$
$X_3 : \mathbb{P}\, SEAT \rightarrowtail states\, X$	$maxreq(T_1, timeout) = 10$
$S_1 : MACHINE \rightarrowtail states\, S$	$maxreq(X_2, timeout) = 0$
$enquire : \mathbb{P}\, SEAT \rightarrowtail EVENT$	$\forall s : range\ X_3 \bullet maxreq(s, timeout) = 0$
$book : NAME \times SEAT \rightarrowtail EVENT$	$\forall s : range\ X_4 \bullet maxreq(s, timeout) = 0$
$get,\ rel : MACHINE \rightarrowtail EVENT$	$\forall s : range\ X_5 \bullet maxreq(s, timeout) = 0$
$in_1 : SEAT \rightarrowtail EVENT$	
$in_2 : NAME \rightarrowtail EVENT$	

Figure 8.12: Seat booking, with mutual exclusion & time out

Chapter 9

Conclusions and Recommendations

In this project, our aim has been to develop software tools and formal notations that facilitate software development in the FORTH programming environment with a particular interest in "Real-Time" and safety critical systems. In this chapter, we provide an overview of the project, linking the various parts, giving comments on the work and making suggestions for possible further work.

9.1 Introduction

Our initial areas of investigation were:

- To provide a method of communicating between multiple FORTH-based RISC systems.
- A version of the FORTH++ compiler to operate with the Harris RTX–2000 stack based RISC processor. This compiler also includes an extensive native-code optimisation technique.
- An interface between FORTH and a local area network. This is to provide a multi-platform development environment. The interface can also be used to pass messages between different (possibly remote) nodes.
- The interfaces between FORTH and other high-level languages, allowing the developer the freedom to interface with supplier propriety code. This was done in such a way that FORTH's interactive user interface is maintained and may be used to enhance the original (non interactive) system.

We have been able to address some of these problems directly. We have described these in various sections: Communicating Novix NC4016s (Appendix A); FORTH++ and the Harris RTX–2000 (Appendix B); Using IBM's NETBIOS from FORTH (Chapter 2); Mixed Language Interface (Chapter 3 and Appendix C).

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 an underpinning. This work mainly consisted of:

- We 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 (Chapter 4).
- In defining our formal base we found a relationship between the stack based virtual machine and a register based target processor. We investigated this relationship, developing a single pass optimisation method (Chapter 5).

- When using our formal definition we found that the parameter stack was a source of uncertainty. The stack is type free, yet to formally define the meaning of a routine we must know the type of the stack items. We therefore investigated the possibility of developing a typed version of FORTH, developing a "type signature algebra" that allows us to type check the FORTH parameter stack (Chapter 6).

  Having defined a type signature algebra we then investigated how this may be implemented and what effect it may have on a program (Chapter 7).

- In order to support the multi-tasking capabilities of FORTH we developed a (formal) theory of concurrent tasks based on state machines that synchronise on events.

  As this system is to be used by people unfamiliar with formal notations we have provided a graphical representation of the theory (Chapter 8).

9.2 Networks

We investigated the IBM NETBIOS system as a standard for interfacing with LANs. The NETBIOS system is designed to operate in parallel with any application. Chapter 2 describes an interface that we have developed to exploit this ability. We have also developed several demonstration programs to show how this system can be used to pass messages from one IBM PC to another.

9.3 Mixed Languages Interface

Our mixed languages interface is an interface between the FORTH programming environment and another programming language. The system described in Chapter 3 is designed to interface with code written in the C language, however, the method is largely language independent thus allowing the developer to interface with code written in any language. Although the system was written with the Microsoft C compiler in mind, it was developed using Borland's Turbo C compiler. We have tested the "portability" of this system by compiling the same code under a number of different compilers.

This interface has shown how we can tap into the ever growing pool of application libraries available for C (and other languages), thus reducing the maintenance and manufacturing costs of our system while providing more functionality.

We have seen one implementation of the general method of integrating these language features. We can see how the general methods can be used to extend the FORTH system into areas that were out of reach. This system would allow us to indulge in hiding of information that the FORTH system has no requirement to access in the C system, such as the floating point stack in Appendix C.

There are several ways in which this work could be advanced. It should be possible to provide a FORTH to C interface on a remote target system or a Pascal or even an Ada interface based on these ideas. It would also be possible to apply this interface to other FORTH systems with relative ease.

9.4 Formal FORTH

Several aspects of our investigations appear to be dependent on a more thorough theoretical underpinning of the language. Thus we moved our attentions to providing such support. Due to 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. Chapter 4 shows some preliminary investigations in to how formal descriptions can be applied to parts of the FORTH programming environment.

9.5 Stack Optimisation

We investigated the relationship between the FORTH virtual stack machine and a register based host processor. In developing our formal support (Chapter 4) we modelled the stack as a sequence of untyped elements. This has lead to the development of a compiler optimisation technique based on this concept. Chapter 5 reviews the more common optimisation techniques and develop our own "stack registers" technique. This technique keeps the top three elements of the stack in internal registers. Unlike traditional systems that keep the items in specific registers, our system allocates the internal registers *dynamically*. We use a relationship between the top elements of the stack and their allocated register to produce a *stack image*. Using this technique it is possible to obtain 100% optimisation on certain stack manipulation operations (such as `SWAP`, `ROT` and `DROP`).

By combining the most common existing techniques (*inline compilation*, and *peep-hole optimisation*) with our *stack registers* we can provide a very powerful optimising compiler.

9.6 Type Algebra

Traditionally the FORTH parameter stack has always been typeless. This allows the programmer to "cast" data items without recourse to inefficient and unnecessary subroutines (as in C). The majority of FORTH programmers use this ability (either directly or indirectly) in almost all applications. Yet it carries with it the problem that the programmer must keep track of the logical type of all the items he is using. It would be possible to fetch an *integer* from a variable, thinking that it was an *execution-token*, causing the system to crash when this *pseudo execution-token* was executed.

In developing our formal support we found that we had to leave the stack as a sequence of untyped elements. Although we could reason about the logic of a program, we need to know the stack types to be able to check if a program is "correct" or not. For this we would need to provide some form of typing mechanism.

Jaanus Pöial of Tartu University has developed a "Stack Type Algebra" (Pöial 1990) that provides the capability of checking the type of stack elements. Chapter 6 presents a new type algebra based on these ideas. The algebra includes the capability of handling items of variable type in addition to catering for program structures and conditional execution.

Using our type algebra, it would be possible to say that a program handles its stack correctly. It is not possible to say that the program will perform as required. By combining the type algebra with the formal toolset, given in Chapter 4, it should be possible to formally prove that a FORTH program has a given property. In this way, we can prove that the FORTH program has the same properties as the specification and is thus a true implementation of that specification.

9.7 FORTH Type Checker

Having developed our "type algebra" we investigated the possibility of implementing a FORTH type checker based on these ideas. Chapter 7 presents an initial specification for such a program, referred to as "FLINT". The specification also allows the program to check for a number of additional errors (such as stack underflow) and various software metrics in addition to its type checking rôle. Such a system will stop the abuse of the typeless stack whilst maintaining the flexibility of a type free stack.

Although we have specified a possible type checker and some thought to its initial design has been incorporated in this specification, we have not, as yet, developed the system. A commercial vendor has shown an interest in this work and may develop an implementation.

9.8 The Event Calculus

The "Event Calculus" is a diagrammatic notation that provides an easily used means of formally specifying the behaviour of concurrent systems. It can describe synchronous and asynchronous communications, data flow modelling and function application, in addition to temporal constraints. It also abbreviates the description of complex state changes such as data base updates via the use of formal notations. This gives the designer a good level of control over the levels of abstraction used.

Chapter 8 introduces the Calculus in a tutorial like manner, initially introducing the notation then building up its functionality one step at a time. The formal definition of the Calculus is given alongside an informal introduction to the notation.

We have found that the diagrammatic nature of the Calculus makes it relatively easy for non specialist to use when compared to other (event based) process algebras such as CSP, CCS and LOTOS. 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.

The Calculus is used to not only represent the state transitions in one system but also the interaction between different state machines. There is a very simple correlation between a state machine as represented in the Calculus and a FORTH task. Indeed, the Calculus appears to be an effective mechanism for breaking down a complex problem into a multi-tasking implementation.

9.9 Future Directions

In this section we describe how the work presented in this document could be extended. We have identified three areas of interest that we feel worthy of further development.

9.9.1 Type Algebra

The type algebra can be developed further:

- The rules for handling variable types (rules 5–8) are probably not required, they can be derived from the matching rules (rules 1–3) and the reduction rule (rule 4).
- The "Multiple Signature" (+) and "Alternative Type" (|) capabilities are the only way subtypes can be expressed at current. The provision of a types hierarchy should be investigated.
- The implementation of the algebra in some form of automatic tool, such as FLINT, or by incorporating the algebra into an existing compiler.

9.9.2 Formal FORTH

It is our intention to provide a secure FORTH programming environment. We intend to produce such a system on a Windows based workstation, such as a Sun (Sparc station) or an IBM PC under Windows. This system should be a formally specified standard FORTH implementation[1]. The formal specification, probably written in Z, will be based on the preliminary "Formal FORTH" work.

We would like to include type checking facilities based on the type algebra, similar to those prescribed for the FLINT program. In time we would like to extend this system to include a formalised variant of the stack registers optimisation technique.

[1] Probably the ANSI Standard, when it is released.

9.9.3 Event Calculus

The interface between an event and a Z Schema is perhaps more complicated than is necessary. This interface should be investigated, a new less complex one should be provided if possible.

The Event Calculus should be used to generate some example specifications. Taking a number of case studies from initial specification, through to final implementation (in FORTH), complete with corresponding proofs. This will also provide us with a less formal introduction to the Calculus.

Such examples could be used to introduce the FORTH community to formal methods whilst the Event Calculus could be used to introduce the formal methods community to the FORTH language.

Bibliography

Aho, A. V., R. Sethi, and J. D. Ullman (1986). *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley.

Almy, T. (1987). Compiling of FORTH for performance. *Journal of* FORTH *Application and Research 4*(3), 379–388.

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

Astle, R. (1985). Yet another recursive decompiler. In *Proc. FORML Conf.*, San Carlos, CA. FORTH Interest Group.

Bailey, G. V. et al. (1987, February). *Poly*FORTH *ISD-4 NC4000 CPU Supplement*. Manhattan Beach, CA: FORTH Inc.

Borland International (1988). *Turbo C Reference Guide*. Scotts Valley, CA: Borland International. Version 2.

Borland International Inc. (1988). *Turbo C User's Guide*. Scotts Valley, CA: Borland International Inc. Version 2.

Bowen, J. P. (1987, January). The formal specification of a microprocessor instruction set. Technical Monograph PRG-60, Oxford University Computing Laboratory, Oxford.

Bradley, M. (1985). Self-understanding programs. In *Proc. FORML Conf.*, San Carlos, CA. FORTH Interest Group.

Bradley, M. and M. Saari (1988). *Sun* FORTH *User's Guide*. Mountain View, CA: Bradley FORTHware.

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.

Bruno, J. and T. Lassagne (1975). The generation of optimal code for stack machines. *Journal of the ACM 22*(3), 382–396.

Buege, B. (1984). A decompiler design. In *Proc. FORML Conf.*, San Carlos, CA. FORTH Interest Group.

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.

Cook, R. and I. Lee (1980, 26–28 May). An extensible stack-oriented architecture for a high-level language machine. In *Proc. of the International Workshop of High-Level Language Computer Architecture*, Fort Lauderdale, FL, pp. 231–237.

Danile, P. and C. Malinowski (1987, June). FORTH processor core for integrated 16-bit systems. *VLSI Systems Design 8*(7), 98–104.

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

Dowling, T. (1981). Automatic code generator for FORTH. In *Proc. FORML Conf.*, San Carlos, CA. FORTH Interest Group.

Duff, C. B. (1984). NEON — extending FORTH in new directions. In *Proc. FORML Conf.*, San Carlos, CA. FORTH Interest Group.

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.

Duncan, R. et al. (1988). *The MS-DOS Encyclopedia*. Redmond, WA: Microsoft Press.

Forth Interest Group (1980). FORTH-*79 Standard*. San Carlos, CA: Forth Interest Group.

Forth Interest Group (1983). FORTH-*83 Standard*. San Carlos, CA: Forth Interest Group.

Glass, B. (1989, January). Understanding NETBIOS. *Byte*, 301–306.

Golden, J., C. Moore, and L. Brodie (1985, March). Fast processor chip takes its instructions directly from FORTH. *Electronic Design*, 127–138.

Hand, T. (1988). Software metrics for FORTH. In *Proc. Rochester* FORTH *Conf. on Programming Environments*, Rochester, NY, pp. 67–68. Institute of Applied FORTH Research.

Hanson, D. R. (1980). Code improvement via lazy evaluation. *Information Processing Letters 11*(4–5), 163–167.

Harris, K. R. (1985). Analyzing large FORTH programs by using the STRUCTURE-TOOL program. In *Proc. FORML Conf.*, San Carlos, CA. FORTH Interest Group.

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

Harris Semiconductor (1988b). *RTX 2000 Instruction Set*. Melbourne, FL: Harris Corporation.

Harris Semiconductor (1988c). *RTX 2000 Real Time Express Microcontroller Data Sheet*. Melbourne, FL: Harris Corporation.

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.

Hoffmann, U. (1991). Stack checking — a debugging aid. In *Proc. EuroFORML Conf.*, San Carlos, CA. FORTH Interest Group.

IBM Corporation (1987, April). NETBIOS *Application Development Guide*. IBM Corporation.

Int$_e$l Corporation (1981, August). *iAPX 86,88 User's Manual.* Berkeley, CA: Int$_e$l Corporation.

Jennings, E. (1985, October). The novix NC4000 project. *Computer Language 2*(10), 37–46.

Jones, T., C. Malinowski, and S. Zepp (1987, May). Standard-cell CPU toolkit crafts potent processors. *Electronic Design 35*(12), 93–101.

Kavipurapu, K. and H. Cragon (1980, 26–28 May). Quest for an 'ideal' machine language. In *Proc. of the International Workshop of High-Level Language Computer Architecture*, Fort Lauderdale, FL, pp. 33–39.

Kernighan, B. W. and D. M. Ritchie (1988). *The C Programming Language* (Second ed.). Software. Englewood Cliffs, NJ: Prentice Hall International.

Knaggs, P. J. and W. J. Stoddart (1990). The FORTH++ 'C' interface. In *Proc. FORML Conf., Proc. EuroFORML Conf.*, San Carlos, CA. FORTH Interest Group.

Knaggs, P. J. and W. J. Stoddart (1991a). The cell type. In *Proc. Rochester* FORTH *Conf. on Automated Instruments*, Rochester, NY, pp. 55–57. Institute of Applied FORTH Research.

Knaggs, P. J. and W. J. Stoddart (1991b). Formal FORTH. In *Proc. Rochester* FORTH *Conf. on Automated Instruments*, Rochester, NY, pp. 50–55. Institute of Applied FORTH Research.

Knecht, K. (1982). *Introduction to* FORTH. Indiana: Howard Sams & Co.

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

Koopman, P. J. (1989). *Stack Computers: The New Wave.* Chichester: Ellis Horwood.

Lukasiewicz, J. (1963). *Elements of Mathematical Logic* (Second ed.), Volume 31 of *International Series of Monographs in Pure and Applied Mathematics.* Oxford: Pergamon Press Ltd. First published in 1929.

Lyons, G. B. (1980). Compressed FORTH object code. In *Proc. FORML Conf.*, San Carlos, CA. FORTH Interest Group.

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

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

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

Microsoft Corporation (1984–85). *Microsoft Macro Assembler.* Redmond, WA: Microsoft Corporation. Version 4.0.

Microsoft Corporation (1984–87a). *Microsoft C Language reference.* Redmond, WA: Microsoft Corporation. Version 5.

Microsoft Corporation (1984–87b). *Microsoft C Run-Time Library Reference.* Redmond, WA: Microsoft Corporation. Version 5.

Microsoft Corporation (1984–87c). *Microsoft C Users Guide.* Redmond, WA: Microsoft Corporation. Version 5.

Microsoft Corporation (1984–87d). *Mixed Language Guide.* Redmond, WA: Microsoft Corporation.

Microsoft Corporation (1987). *Code View and Utilities.* Redmond, WA: Microsoft Corporation.

Miller, D. (1987, April). Stack machines and compiler design. *Byte 12*(4), 177–185.

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

Minker, J. and R. G. Minker (1980). Oprimization of boolean expressions — historical developments. *A. of the History of Computing 2*(3), 227–238.

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.

Moore, C. (1990a). *MuP20 Microprocessor: Preliminary Specifications*. Woodside, CA: Computer Cowboys.

Moore, C. (1990b). ShBoom on ShBoom: A microcosm of software and hardware tools. In *Proc. Rochester* FORTH *Conf. on Embedded Systems*, Rochester, NY, pp. 21–27. Institute of Applied FORTH Research.

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

Morse, S. P. (1982). *The 8086/8088 Primer* (Second ed.). Chichester: Hayden Book Company.

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

Nine Tiles (1987, September). *Superlink Reference Manual*. Cambridge, UK: Nine Tiles.

Nine Tiles (1988a, February). *SimpleNet User Guide*. Cambridge, UK: Nine Tiles.

Nine Tiles (1988b, June). *SimpleNetBIOS Reference Guide*. Cambridge, UK: Nine Tiles.

Novix (1985). *Programmers' Introduction to the NC4000 Microprocessor*. Cupertino, CA: Novix Inc.

Pawley, W. (1984). Using native machine code analogs of interpreted FORTH's elements for high performance. In *Proc. Rochester* FORTH *Conf. on Real Time Systems*, Rochester, NY, pp. 115–120. Institute of Applied FORTH Research.

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.

Pöial, J. (1991). Multiple stack effects of FORTH programs. In *Proc. EuroFORML Conf.*, San Carlos, CA, pp. 400–406. FORTH Interest Group.

Rather, E. D. (1985). FORTH. *Computer Programming Management*.

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

Rather, E. D., L. Brodie, et al. (1986, November). *Poly*FORTH *ISD-4 Reference Manual* (fifth ed.). Manhattan Beach, CA: FORTH Inc.

Rose, A. (1986). Design of a fast 68000-based subroutine threaded FORTH with inline code & optimiser. In *Proc. Rochester* FORTH *Conf. on Artifical Intelligence*, Rochester, NY, pp. 285–288. Institute of Applied FORTH Research.

Schwaderer, W. D. (1988, August). *C Programmer's Guide to NetBIOS.* Indianapolis: Howard W. Sams & Company.

Scott, A. (1989). An extensible optimizer for compiling FORTH. In *Proc. FORML Conf.*, San Carlos, CA. FORTH Interest Group.

Silicon Composers (1987a, March). *PC4000 Board User Manual.* Palo Alto, CA: Silicon Composers.

Silicon Composers (1987b, January). *SC*FORTH *User Manual* (second ed.). Palo Alto, CA: Silicon Composers.

Sjolander, S. (1987). Novix decoder. In *Proc. FORML Conf.*, San Carlos, CA. FORTH Interest Group.

Sjolander, S. et al. (1987, June). *Poly*FORTH *ISD-4 Intel 8086 CPU Supplement* (ninth ed.). Manhattan Beach, CA: FORTH Inc.

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

Spivey, J. M. (1990, September). Specifying a real-time kernel. *IEEE Software*, 21–28.

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

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. (1987). Nested error handlers. *Journal of* FORTH *Application and Research 4*(3), 443–445.

Stoddart, W. J. (1988). Specification & optimisation. In *Proc. EuroFORML Conf.*, Southampton. MicroProcessor Engineering Ltd.

Stoddart, W. J. (1989a). FORTH++ *Course Notes — Part 1: An introduction to* FORTH. Middlesbrough, UK: Teesside Polytechnic.

Stoddart, W. J. (1989b). FORTH++ *System Documentation.* Middlesbrough, UK: Teesside Polytechnic.

Stoddart, W. J. (1990a). FORTH++ *Course Notes — Part 2: Multi-tasking and Windows.* Middlesbrough, UK: Teesside Polytechnic.

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

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

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

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

Stoddart, W. J. (1991a). FORTH++ *C/*FORTH *Interface.* Middlesbrough, UK: Teesside Polytechnic.

Stoddart, W. J. (1991b). FORTH++ *Extras: Graphics support and software Floating point.* Middlesbrough, UK: Teesside Polytechnic.

Stoddart, W. J. and P. J. Knaggs (1990). FORTH++ and the MACH1 RTX-2000 board. In *Proc. FORML Conf., Proc. EuroFORML Conf.*, San Carlos, CA. FORTH Interest Group.

Stoddart, W. J. and P. J. Knaggs (1991). Type inference in stack based languages. In *Proc. Euro-FORML Conf.*, San Carlos, CA. FORTH Interest Group.

Stoddart, W. J. and P. J. Knaggs (1992, September). The Event Calculus (Formal Specification of Real Time Systems by means of Diagrams and Z Schemas). In *Proc.* 5^{th} *International Conf. on Putting into practice methods and tools for information system design.* University of Nantes, Institut Universitaire de Technologie, 3 Rue du Maréchal Joffre, 44041 NANTES Cedex 01, France.

Surrey Medical Imaging Systems Ltd. (1988). *PP2000 50 MIP parallel processor RTX-2000* FORTH *development board.* Guildford, UK: Surrey Medical Imaging Systems Ltd.

Tanenbaum, A. S., H. van Staveren, and J. W. Stevenson (1982). Using peephole optimization on intermediate code. *TOPLAS 4*(1), 21–36.

Wickes, W. C. (1988). RPL: A mathematical control language. In *Proc. Rochester* FORTH *Conf. on Programming Environments*, Rochester, NY, pp. 27–32. Institute of Applied FORTH Research.

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

Zorland Inc. (1987). *Zorland C Compiler.* Zorland Inc. Version 2.

ZorTech Inc. (1989a). *ZorTech C++ Compiler Compiler Reference.* ZorTech Inc. Version 2.

ZorTech Inc. (1989b). *ZorTech C++ Compiler Function Reference.* ZorTech Inc. Version 2.

Appendix A

Communicating Novix NC4016s

At the start of this project we were given two Novix NC4016 boards. The intention being to work with these boards to provide systems that could communicate over multiple CPUs. The Novix NC4016 being a RISC based processor that is capable of executing a FORTH program at great speed.

A.1 Introduction

The Novix processor is sufficiently small that it takes only 4000 gates in a programmable logic array to implement. The chip has four different data paths, it is possible to use all four within one machine instruction, thus providing the capability of executing up to five FORTH instructions in one machine cycle. The majority of machine instructions are executed in one clock cycle. As a result a Novix system operating at 10 MHz has a peek effective throughput of 40 MHz or 40 MIPS (Million Instructions Per Second).

The machine instructions of the Novix are designed to execute FORTH. The FORTH systems supplied with the boards were designed to compile optimised native code for the Novix system.

In this annex we describe the Novix plug in boards in detail and describe how we made the two totally independent boards communicate with each other.

A.2 Programming

Four different implementations of the FORTH programming language were provided:

A.2.1 cmFORTH

This is an implementation by Chuck Moore. It is a basic system that has been modified to exploit the abilities of the Novix. We found the FORTH itself, whilst a valid implementation, to be lacking in features[1]. This is used in conjunction with the *SC*FORTH system to provide a programming environment on the Novix board.

A.2.2 *SC*FORTH

This is a system provided by Silicon Composers to operate on the IBM PC. It is a rather basic implementation of the language that had been extended to allow communication with the Novix boards. Although the implementation is valid, it was of a rather basic system with few features.

[1] This is a common feature of all of Chuck's FORTH systems.

To use the Novix board, the user would load the *SC*FORTH system. He would then load the **cm**FORTH system into a 16 KBytes dual ported memory buffer. The *SC*FORTH would then produce an interrupt signal that instructs the Novix to start executing the **cm**FORTH system. This provides a programming environment on the Novix system and a support environment on the IBM PC.

Having set the Novix executing the **cm**FORTH system, and any application the user may have loaded, he can then exit from the *SC*FORTH system back to the IBM PC's operating system. Thus leaving the Novix system to continue executing its application. This is only valid if the application does not require any facilities from the IBM PC "Host" (see section A.4), as these facilities have now been removed.

A.2.3 *Poly*FORTH

This is a fully implemented FORTH development system. Two versions of the system are provided. One operated on the IBM PC while the other operated on the Novix. This is a full FORTH system with a full screen editor, multi-tasking and meta-compilation. The system also provided a target compilation system for use with the Novix.

The Novix *Poly*FORTH system is also a full development system. It has multi-tasking, meta-compilation and optimisation. As with the Silicon Composers system, the user has to load the *Poly*FORTH system. This will then load the Novix *Poly*FORTH system into the dual ported memory and set the Novix executing it. The IBM PC system provides "Host" services for the Novix system.

As with the Silicon Composers system, once the Novix system has been loaded and is executing an application, the IBM PC's *Poly*FORTH system may be removed providing that the Novix application does not require any of the "Host" facilities provided by the IBM PC software.

A.2.4 FATWIN

The "*F*ORTH with *A*rgument records, Multi-*T*asking and *Win*dows" system developed by Bill Stoddart was supplied as a FORTH system that operated on the IBM PC. This is a full implementation of FORTH with many standard extensions and several non-standard features. Most of the extensions on this system are provided in a manner that is compatible with the *Poly*FORTH implementation.

Whilst this system was of little interest when the project began it was latter developed into the FORTH++ system and has played a large part in this project (see chapter 1).

A.3 Single Boards

We were using the *Poly*FORTH system to develop the multi-processor communications systems. This operated correctly with the first of the two boards but not with the second. The location of the dual ported memory of the first board was at segment `CC00` and segment `D000` for the second.

All of the software in the IBM PC *Poly*FORTH system used the variable `'FB` to hold the segment address of the dual ported memory. The software in the Novix *Poly*FORTH system does not need to know where this memory is located in the IBM PC, it simply uses the memory at `00000` though to `04000` as its communication area.

Thus we can change the board that the *Poly*FORTH system is communicating with simply by changing the value stored in the `'FB` variable. We defined two new words `FB1` and `FB2` to set the value of `'FB` to communicate with the first or second board as required. As the *Poly*FORTH system only copies the Novix *Poly*FORTH to the first board it is left to the user to "boot" the second board. This is done by selecting the second board and typing the FORTH word `BOOT`:

```
FB2 BOOT
```

Whilst this method of booting the second board works, we want to hide such information from the user. Thus, we redeveloped the code for `FB1`, `FB2`, `BOOT` (and `PLUG`) so that when the user selected a

board that had not been initialised, the system would automatically copy the *Poly*FORTH system to the new board and set it operating. The code that created the words `FB1` and `FB2` was developed in such a way that the user can add new boards to the system at will. The code is given in section A.7.1.

A.4 Host Services

When the FORTH word `PLUG` is executed the *Poly*FORTH system connects with the current Novix board (indicated by the value of `'FB`). The `OPERATOR` task is set up to provide "Host Services" to the Novix system. The purpose of this is to provide the Novix system with a method of accessing the IBM PC's hardware. The *Poly*FORTH system provides access to the Keyboard, Video, Disks, Communication Ports and Printer.

The mechanism by which these services are provided is that of a "Mailbox". One mailbox is used for each device that the system is supporting. The mailboxes are stored in the dual ported memory, thus being accessible by both the Novix and the IBM PC. The mailboxes are of differing length depending on the device that they are controlling and the functions it can perform.

The Novix will place a value in the first word of the mailbox indicating the function it would like to perform. The IBM PC will take this value and use it as an offset into a table of functions[2] executing the required function. The IBM PC will place the value -1 in the mailbox to indicate to the Novix that it has completed the requested function. Whilst this is not a very good FORTH based approach to the problem, it is the way the original *Poly*FORTH system operated. In our work, we have tried not to alter the way in which the Novix communicates with the Host in any way thus we have inherited this message passing mechanism.

For the Novix to display a message on the IBM PC's screen, it would place the required text into the buffer area of the mailbox. It will then set a function code in the mailbox and wait for the IBM PC to acknowledge the function code. The IBM PC will interpret the function code and display the text in the buffer. Having completed the function it will send a message back to the Novix in the form of an acknowledgment. Having received the acknowledgment the Novix will now continue with its processing.

The system is designed in this way in order to provide multi-tasking on the Novix board. Let us say, for example, that two independent tasks on the Novix would like to access the same device. The first task would place its function code into the mailbox (and additional data if required). This task would then continue processing. However, the second task would see that the mailbox has a function pending (ie, that the function request code is not -1) and so will wait (in a multi-tasking manner) until the IBM PC acknowledges the first task request by placing -1 in the request code area. The second task can now continue and place its request in the mailbox.

The *Poly*FORTH "Host" system has a task constantly monitoring the function request value for each mailbox, with the exception of the Keyboard and Video mailboxes. When the IBM PC is `PLUG`ed into the Novix board the `OPERATOR` task monitors the Keyboard and Video mailboxes. One of the functions associated with this mailbox is "return to host". On receiving this request the `OPERATOR` reverts back to its normal action of interpreting keyboard commands for the IBM PC system.

A.5 Parallel Boards

The system described in sections A.3 and A.4 provides us with a mechanism that allows two (or more) separate Novix boards operating in parallel. There are still two problems to be solved:

1. Only one Novix board may have access to the "Host" facilities at any given time. This access is controlled by the operator connecting with the relevant board (via the use of `FB1` and `FB2`).

2. Although the boards may be operating in parallel, they are also operating in isolation. A method that allows the boards to communicate with each other is to be found.

[2] This means that the function numbers must be even.

We see two methods of solving the first of these problems. Namely the provision of "Host" services to all of the Novix systems available.

A.5.1 First Method

The monitoring of the mailbox function request area is performed by the word `AWAIT`. The code for `AWAIT` was redeveloped to scan for both Novix systems (see section A.7.2). Thus when the monitoring task executed the `AWAIT` word it would automatically monitor both of the Novix boards. When `AWAIT` returned the system would be ready to perform the required action for the correct board.

This system failed to operate to our expectations for various reasons:

1. The system is based around the ability of altering the value of the `'FB` variable to indicate the relevant Novix board. This is a *global* variable, thus, changing its value, will effect other tasks. As the system is multi-tasking, it is not possible to change such a variable and expect its value to be the same when next used.

   This is a common problem when dealing with multi-tasking systems. By changing the `'FB` variable from a *global* to a *user* variable, each task will have its own copy of the variable. Hence it can take on different values for each different task.

2. The code (as given in section A.7.2) is explicitly written for use on two boards and is not capable of expansion (without rewriting it each time a new board is added).

   By changing `MAKE.FB` we could make a cyclic list out of the possible Novix boards present. We are then able to rewrite `AWAIT` to scan through the cyclic list of possible boards. Hence, when the user creates another board, it would automatically be added to the list of boards. Thus, the monitoring software will automatically start servicing it.

   This has a problem in that the host will start servicing the board before it has been initialised. By moving the code that links the new board into this cyclic list to be the last function performed by the `FBOOT` word we stop the host from servicing the new board until after it has been initialised. The code for this system is given in section A.7.3.

3. The `DISKER` task provides access to the disk system, via use of the disk mailbox. This is a short mailbox holding two data fields, the value `#BLK` indicating which disk block the Novix requires and the value of `BUFFER` indicating whereabouts in the IBM PC's memory the block has been located. To read the contents of the disk block, the Novix issues keyboard input requests. The keyboard server will take its input from the indicated block rather than the keyboard.

   Whilst this is (in principle) the correct way of performing this function[3] it has two problems:

   (a) It relies on the interaction of two servicing tasks, the `DISKER` task (monitoring the disk mailbox) and the `OPERATOR` task (monitoring the keyboard/video mailbox). It is possible for the `OPERATOR` to be processing another board's request at the same time the `DISKER` request is made. However, the `OPERATOR` will see the flag indicating it should take input data from the disk block and will therefore pass the disk block information to the wrong Novix system.

   (b) It means that the Novix system can only process disk information when the keyboard/video mailbox server is in operation. Ie, the board in question is connected.

   If we were to "tighten up" the synchronisation between the two tasks, so that the `OPERATOR` task were to know for which Novix board the disk request applies, it would stop the first problem. However, the Novix system would still only be able to access the disk services when it has been connected too.

[3] Based on the description of disk accessing given in the Forth'83 standard.

4. In order to communicate with the boards the user must connect with one of them (using the `FB1` or `FB2` words). This will set the `OPERATOR` task servicing the keyboard/video mailbox. However, as the keyboard is being treated as a normal device two (or more) Novix system may make a request to obtain input from the keyboard. One of the systems will be honoured, however, the user will have no knowledge as to which system he is communicating. This relates to the previous problem of the disk accessing being directed through the keyboard/video mailbox.

   By providing an additional task `TERM` that will service most of the keyboard/video mailbox requests, we can provide all of the boards with access to the video. We can also provide access to the disk system by synchronising the `TERM` task with the disk accessing of the `DISKER` task. This will leave the `OPERATOR` task to be used in the conventional manner.

   The `TERM` task will be able to process requests for text from the keyboard under two circumstances:

   (a) The Novix board making the request has previously made a request of the `DISKER` task (ie, is inputting a disk block).

   (b) The `OPERATOR` task had been directed to operate as the keyboard for the Novix system making the request.

   In this manner, we can provide a mechanism that allows any of the Novix boards to access the IBM PC's disk drives. The user will be able to `PLUG` into any of the boards and know that all keyboard communication is being directed to that board.

   This solution still has a problem. When the user is connected to one of the boards, it will stop any form of disk accessing. The Novix will issue a "Read data from the keyboard" request. While the `TERM` task is processing this request it will not be able to process any other requests. The solution to this problem is to totally redevelop the functions provided by the disk mailbox such that it operates independently of the keyboard mailbox.

A.5.2 Second Method

The second method of provided this facility is to totally redevelop the servicing code. By providing a single task that scanned and processed all of the mailboxes connected with the one board it is possible to add as many tasks as required, each servicing its own Novix system.

Whilst this method is conceptually simpler than the first, it means a complete redevelopment of the existing code, the converging of several tasks into one. Although this made the problems of handling the disk and keyboard into simple problems.

A.5.3 Comparison

Both methods provided the environment that is required. The first method is more difficult to understand and leads to complicated problems that were solved by redeveloping part of the server code (and the Novix code). It is simpler to use, as it is more automatic than the second method.

The second method is simpler to understand but requires a great deal more effort to totally redevelop the servicing code, although no Novix code has to be changed. The coding has to take into account the fact that several tasks may be requesting access to the same resource simultaneously whilst, in the first method, this is taken care of automatically (by the logical splitting of tasks).

The second method is more open to adaptability than the first. With the second method it is possible to use two entirely different FORTH systems on the Novix boards, each having its own servicing task on the IBM PC. For instance, it is possible to run *Poly*FORTH on the first Novix board and **cm**FORTH on the second (provided that a servicing task has been developed to operate with the **cm**FORTH system). The first method is simply not easily extendible.

A.6 Communicating Systems

Section A.5 describes a system that allows any number of Novix boards to operate in parallel, accessing host services as required. The user may communicate with one board at a time from the keyboard. Although the boards may be operating in parallel, they are still unable to communicate with anything except for the host. In this section, we extend this system to allow different Novix boards to communicate with each other without the intervention of the user.

A.6.1 Hardware Restrictions

The main problem in providing such a facility is the Hardware Restrictions inherent in the design of the board. The boards provide a 16 KBytes area of dual ported memory that is mapped into the IBM PC's main memory area thus allowing the IBM PC to access the first 16 KBytes of the Novix memory. The exact location of this memory can be configured (in the IBM PC) by setting jump switches on the Novix board. No two Novix boards may have a configuration using the same memory area on the IBM PC for their dual ported memory as this would cause a "bus error", in addition to being a logical error. No two dual ported memory areas may overlap as this will also cause a "bus error".

Thus, for each Novix board in use this 16 KBytes of dual ported memory are physically separated (by being located on the Novix boards) and must be logically separated (by being mapped into different areas of the IBM PC's memory).

A.6.2 Communication

Given that the hardware design inhibits any form of inter-processor communication (except with the IBM PC), a software solution is to be found.

As the IBM PC is the only system that is able to communicate with all of the boards present, we can consider a facility to allow communication between boards to be an additional host service. As we have already modified the way in which the host services are provided we are able to extend the system to include inter-processor communication (between Novixs). For further information on host services see section A.4, and section A.5 for more information on their modification to allow multiple board operation.

Mailbox

We have provided an additional mailbox for "Inter-Processor Communication" (IPC). The mailbox has the format given in figure A.1.

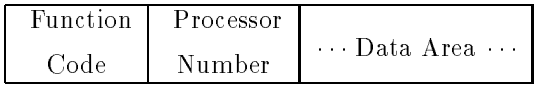


Figure A.1: Format of the IPC Mailbox

Where *Function code* is the function request code area, *Processor number* is a special data area used by all of the available functions and the *Data area* is where the IPC message is placed. The system we provide does not make any checks on the data area. The message may be in any format, as required by the application.

Identification

A system of identifying processors is required. There are basically three different ways of identifying a processor:

1. The segment address of its dual ported memory area can be given. This would seem the most useful option as it requires less code and is quick in operation. However, it means that each existing Novix will need to know the configuration of the other Novix's dual ported memory.

2. The address of its data area (in the cyclic list). This still has the problem that every Novix will have to have knowledge about every other Novix. This will also add a level of indirection to the system, thus slowing it down. In addition these addresses are likely to change as new code is developed.

3. An index value that indicates the Novix's position in the system. Thus the first system initialised is referred to as processor 1, the second as processor 2 etc. This requires two levels of indirection with extra processing by the IBM PC, however, it also means that the Novix systems needs no knowledge of the other Novix systems. Indeed it will also allow on-line reconfiguration of the Novix systems available.

   It would also be possible for an application to send messages using this IPC system between the IBM PC and the Novixs using this method. In such a case the IBM PC's identification number is given to be 0.

Functions

There are three functions associated with the IPC system:

Who am I? This function places the processor identification number of the calling Novix system in the *processor number* area of the mailbox, thus providing the method for a Novix to discover what its identification number is.

The number of Novix systems currently accessible to the host is placed in the *data area* of the mailbox. This is provided so that application software may configure itself to use all of the Novix systems available to the best advantage.

Wait will wait for a message being directed to this board. A task will be set up on the Novix system that waits for a message from another board. On receiving a message, it will interpret it and act on it as required. The processor identification number of the sending processor is placed in the *processor number* while the message is placed in the *data area*.

Message sends a message to another system. The message to be sent is placed in the *data area* while the processor identification number of the destination processor is in the *processor number* area.

The exact form of the message is application dependent. You may transfer data, in any form, from one processor to another (including the host). The host's message passing service simply copies the data area of the message buffer from one mailbox to the other. The target system must be waiting for a message otherwise an error code is returned (a -1 is placed in the "processor number" area).

As a board can discover its own processor number (via the *Who am I?* IPC function), it would be possible for a multi-tasking application to post a message to another task on the same board by placing its own processor id into the *processor number* field.

A.7 Code

In this section we present the code that was developed during this project. The code is presented in the order it was written with the multi-board boot code first followed by the first attempt at providing host services for multiple boards.

This code is explicitly designed for use with two boards. The redeveloped (true multi-board) version of the code is given in the final section.

A.7.1 Multiple Boards "Boot" code

We start by defining the new version of the two words `PLUG` and `BOOT`. Finally, we define the word `MAKE.FB`. We tell the system that a new board is present by defining a new data area for it with the `MAKE.FB` word. This defines a new word and defines a data area for the new board.

When the new word is executed (the board is referenced) for the first time it will load the *Poly*FORTH system into the board's dual port memory and start the board executing. `MAKE.FB` configures the new word's data area to execute the `FBOOT` word to perform this task.

The last action of the `FBOOT` word is to alter the action of the new word, so that next time it is executed, the system will execute the `FPLUG` word, thus given access to the given board.

We define a new version of `PLUG` that takes the address of the boards data area off the stack and ignores it.

```
: FPLUG ( addr -- ; Version of PLUG invoked by MAKE.FB words )
  DROP            \ Ignore data area address
  PLUG            \ Plug into the given board
;
```

Next, we define a new version of `BOOT` that takes the address of the boards data area and boots the given board. It will then change the value of the board's data area such that the next time the word is executed it will invoke the `FPLUG` word rather than the `FBOOT` word.

```
: FBOOT ( addr -- ; Version of BOOT invoked by MAKE.FB words )

  BOOT            \ Copy PolyFORTH to the Novix board

  ['] FPLUG       \ Get the execution token for FPLUG
  !               \ Alter the data area of the MAKE.FB word

  PLUG            \ Connect with the Novix board
;
```

Now we can define the `MAKE.FB` word. This is a creating word, it will create a named reference for the board, the dual ported memory of which is located at the given segment address.

When the new word is executed the system will connect with the given board, booting it, if the board has not been accessed before.

```
: MAKE.FB ( seg --   ; Create Novix Board access word with
                       dual ported memory at seg )

  CREATE              \ Create the new word

  ,                   \ Its body has the segment address of the dual
                      \     port memory for the given board

  ['] FBOOT ,         \ And the execution token of the FBOOT word

  DOES> ( addr -- )   \ When executed the new word will:

        DUP @         \ Fetch the dual port memory segment address
        'FB !         \ Store it in 'FB

        2+            \ Move to the execution token
        DUP           \ Get the execution token
        @EXECUTE      \ Execute the word (FBOOT or FPLUG)
;
```

It should be noted that when the **@EXECUTE** is executed the **'FB** variable has been set to the correct value for the board in question. The word **FBOOT** is passed the address of the execution token in the word's data area, this is so that it may change it. However, when changed the word **FPLUG** is also passed this value, so we must ignore it.

Finally we make the two boards known to the system.

```
HEX               \ input in Hexadecimal

CC00 MAKE.FB FB1  \ Define FB1 to connect with the first board,
                  \    with its dual ported memory at segment CC00

D000 MAKE.FB FB2  \ Define FB2 to connect to the second board,
                  \    with its dual ported memory at segment D000

DECIMAL           \ revert input back to Decimal
```

A.7.2 First attempt at providing Host Services

This section provides the code for the first attempt at providing host services for both boards (as described in section A.5.1). We start by defining two constants to hold the values of the dual ported memory for the different boards:

```
HEX                \ Numbers are in Hexadecimal

CC00 CONSTANT 'FB1 \ The first board is at segment address CC00
D000 CONSTANT 'FB2 \ The second is at D000

DECIMAL            \ Back to Decimal input
```

Now we redefine the **AWAIT** function such that it looks in the mailbox for both Novix boards alternatively. Examining the mailbox of the first board. If that is -1 (no function) it will then look at the mailbox of the second board, etc. When the value is not -1 there is a function pending. The **AWAIT** word will return with the function number. This matches the behaviour of the original **AWAIT** except for searching both mailboxes.

```
: AWAIT ( mailbox -- n ; Watch mailbox for function from Novix boards )
  -1                  \ Put a dummy function code on the stack

  BEGIN
    DROP              \ Drop the old function code (-1)
    PAUSE             \ Allow other tasks to run

    'FB @ 'FB1 =      \ Switch boards:
              IF 'FB2 \    If 'FB is pointing to the first board
            ELSE 'FB1 \       then set it to the second board
            THEN 'FB ! \      else set it to the first board

    DUP
    request FB@ DUP   \ Get mailbox function code
    -1 = NOT          \ True if function code <> -1

  UNTIL               \ Repeat loop until the function code <> -1

  NIP                 \ Remove mailbox address from the stack
;
```

A.7.3 Revised Boot and Host code

The code given in sections A.7.1 and A.7.2 is described in section A.5.1. This section also gives the problems encountered with this code and the modifications needed to overcome these problems. In this section we present this modified code.

In this code, we provide a cyclic list of Novix board data areas thus allowing us to access as many boards as are available. Otherwise, the code is similar to the code already presented.

In order to handle the cyclic list, we define a global variable (`'FB.HEAD`) to hold the address of the first entry in this list. We initially store a `0` in this variable, thus a `0` value is used to indicate an empty list. In order to overcome the problem of indefinite postponement, we also define the `'FB.LIST` user variable[4] (see the description of the `AWAIT` word on page 106 for more information).

```
VARIABLE 'FB.HEAD      0 'FB.HEAD !
   USER* 'FB.LIST      0 'FB.LIST !
```

We now redefine the `'FB` variable, changing it from a global variable into a user variable, thus each task will have its own copy of the variable.

```
USER* 'FB
```

We now provide the `FPLUG` word that will take the Novix board's data area and ignore it (for the same reason as given in A.7.1).

```
: FPLUG ( addr -- ; Version of PLUG invoked by MAKE.FB words )
  DROP            \ Ignore data area address
  PLUG            \ Plug into the given board
;
```

We now define the `FBOOT` word to work in much the same way as given in section A.7.1. However, having booted the board, it will then add the board to the cyclic list of available boards. If the list is empty it will make this entry link to itself thereby making a cyclic list.

```
: FBOOT ( addr --   ; Version of BOOT invoked by MAKE.FB words )

  BOOT              \ Copy PolyFORTH to the Novix board

  DUP 4+            \ Move to the execution token area
  [']  FPLUG !      \ Alter to the FPLUG execution token

                    \ Add this data area to the cyclic list
  'FB.HEAD @ 0=     \ Is the list empty ?
  IF   DUP !        \ Yes => Make this the last entry of the list
  ELSE DUP          \  No => Copy head of list to this entry
       'FB.HEAD @ !
  THEN 'FB.HEAD !   \ Make this entry the head of the list

  PLUG              \ Connect with the Novix board
;
```

We now define the `MAKE.FB` defining word. The data area holds space for the link required by the cyclic list, the segment address of the dual ported memory and the execution token of the word to execute (`FBOOT` or `FPLUG`) when the new word is executed.

[4] It should be noted that the word `USER*` is a special version of the standard word `USER` which automatically allocates the next free word in the user area.

```
: MAKE.FB ( seg --   ; Create Novix Board access word with
                       dual ported memory at seg )

  CREATE             \ Create the new word

  0 ,                \ Its body has an initial link value of 0
  ,                  \ The segment address of the dual port memory
  ['] FBOOT ,        \ And the execution token of FBOOT

  DOES> ( addr -- )  \ When executed the new word executes:
        DUP 2+ DUP   \ Move to the segment address value
        @ 'FB !      \ Store the boards segment address in 'FB
        2+           \ Move to the execution token
        @EXECUTE     \ Execute the word (FBOOT or FPLUG)
;
```

It should be noted that when the `@EXECUTE` is executed the `'FB` variable has been set to the correct value for the board in question. The memory address of the start of the data area is placed on the stack before invoking the word `FBOOT` or `FPLUG`.

We should now make the two boards known to the system.

```
HEX                \ input in Hexadecimal

CC00 MAKE.FB FB1   \ Define FB1 to connect with the first board,
                   \    with its dual ported memory at segment CC00

D000 MAKE.FB FB2   \ Define FB2 to connect to the second board,
                   \    with its dual ported memory at segment D000

DECIMAL            \ revert input back to Decimal
```

We are now in a position to redefine the `AWAIT` word. This version of the word will look at the mailbox associated with each of the boards in the cyclic list. If the function code is -1, it will move on to the next board in the list, otherwise it returns the function code, setting up the `'FB` variable for communication with the board in question.

In order to overcome the problem of indefinite postponement, the word will always start scanning from the entry indicate by the `'FB.LIST` variable. When a function code is found, a pointer to the next entry in the cyclic list is placed in the `'FB.LIST` variable. The word will start scanning from the next entry in the list when next invoked thus removing the possibility of indefinite postponement.

When the word is first invoked, the value of `'FB.LIST` will be `0` (uninitialised). The word will then check the global head of list (`'FB.HEAD`). If this is also found to be empty, the word will loop until such time as the list has an entry.

```
: AWAIT ( mailbox -- n ; Watch mailbox for a request from Novix boards )

  'FB.LIST @           \ Look at the local start of list

  BEGIN
    0=                 \ Loop if the list is empty
  WHILE
    PAUSE              \ Allow other tasks to run
    'FB.HEAD @         \ Get the global head of list
  REPEAT               \ Repeat until the list is NOT empty

  BEGIN
    PAUSE              \ Allow other tasks to run

    DUP 2+ @           \ Get this board's dual port segment address
    'FB !              \ Set the current board segment address
```

```
    OVER
    request FB@ DUP    \ Get the mailbox request
    -1 =

  WHILE                \ While the mailbox request is -1 (no request)
    DROP @             \ Move on to the next board in the list
  REPEAT

  SWAP @               \ Get the next list entry
  'FB.LIST !           \ Save for scanning next time

  NIP                  \ Drop the mailbox offset
;
```

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).

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

```
: 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.

(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 1990c; Stoddart 1990d; Stoddart 1990e), 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.

```
10 20 30 SURFA .  ENTER  2200 ok
```

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

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 `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`.

```
: $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.

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 1990b) for a full description of 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 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

[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`.

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.

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 1988a). For investigation purposes, we implemented

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

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.

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].

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

[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.

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.

Appendix C

Mixed Languages Interface: Source Code

This annex is intended to supplement the "Mixed Languages Interface" chapter (chapter 3). It gives source listings and some technical comments for the mixed languages interface, as we are currently using it.

The source is split into a number of different files. These files are:

`CFLOAD.ASM` program that loads the FORTH++ system and then the C overlay if it is required.

`MAKELOAD.BAT` batch file that makes the FORTH loader program.

`CFINIT.C` contains the C initialisation code.

`CFORTH.H` header file containing macro definitions (`push`, `pop`, etc.).

`CFORTH1.C` an example "user file". This is the only file that the user should edit. We have provided an example module that provides a floating point maths extension, using the C floating point code.

`CFASM.ASM` holds the code for initialising the C and FORTH context switching area, in addition to the code for performing the context switch.

`MAKEOVLS.BAT, MAKEOVLL.BAT` make the overlay library, incorporating the user supplied code. Using the *S*mall or *L*arge memory models.

C.1 Loader

This is the Microsoft assembler source code for a program that loads in the FORTH++ system. It first returns as much memory as possible to MS-DOS. It will then proceed to load in the FORTH++ segments. For a given segment, the name of the file to load is obtained by taking the name of the loader program and replacing the `.COM` by the required segment extensions:

`.CDE`	Code segment
`.FOR`	FORTH data segment
`.STR`	String segment
`.NAM`	Name segment

The stack segment is grabbed from memory, but is not initialised by this loader program.

After loading in the FORTH system it will then load in a C module. Notice that this module is the last to be loaded. The loader will either take the C file from the same directory as the FORTH++

overlays or from a given location. It will then pass control to the C module to allow it to initialise before executing any of the FORTH code. If no C module is required the loader will simply execute the FORTH directly.

```
        PAGE    60,132  ; Set page size, lines x cols

        TITLE   'CForth++ Loader'

comment ;

This file is assembled to produce the Forth system loader.  The batch
file MAKELOAD.BAT contains the commands to assemble the file and convert
the object code to .COM format.  The file produced by MAKELOAD.BAT is
called FLOADER.TPT.  This file is a loader template which is patched and
renamed by the Forth SAVE-SYS routine to produce a customised loader for
a particular Forth system. The distribution files FPP.COM and CFORTH.COM
are examples of Forth loaders produced in this way.

When a loader such as CFORTH.COM is invoked as a DOS command, it detects
its own name (in this case "CFORTH") and loads in the Forth segments
(CFORTH.CDE, CFORTH.FOR, CFORTH.NAM and CFORTH.STR).  It also reserves
space for the Forth stack segment.  The segment address of each segment
is stored at reserved locations in the Forth code segment.

If a C overlay file was specified at SAVE-SYS time, its name will have
been patched into the loader file and it will be loaded and executed as
an overlay.  This overlay will be passed the PSP of the loader, within
which will be the address of the Forth code segment (in an unused FCB
address in the PSP).

If no C overlay file is specified control will be passed directly to the
Forth code segment.

;================================================================

CFL_TEXT segment byte public 'CODE'
        assume  cs:CFL_TEXT,ds:CFL_TEXT,es:CFL_TEXT,ss:CFL_TEXT

        ORG     100h

entry   PROC    NEAR

; To enable loader locations to remain at a fixed position known to the
; Forth system, the code starts with a jump.

        jmp     start

; To allow a common location to place a debug breakpoint this far return
; instruction is placed at location 103 of the loader.  Hence when the
; Forth word TRAP is invoked, this far return is executed, thus allowing
; us to use a monitor program to assist in the debugging of our Forth
; code.

dbg     PROC    FAR
        ret
dbg     ENDP

; ============================================================================
;                             Data  Area
; ============================================================================
;
; Now we have the data area for the loader.  The default values held in
; this section will be altered to suit by Forth's SAVE-SYS command.
```

```
        ; Forth Segment sizes (in paragraphs)

csize:  dw      1000h    ; Code segment
fsize:  dw      1000h    ; Forth segment
nsize:  dw      1000h    ; Name segment
$size:  dw       800h    ; String segment
ssize:  dw      1000h    ; Stack segment

        ; Far address to execute Forth

eoff:   dw      3        ; Offset within Code segment to start execution
eseg:   dw      ?        ; Will be set when Forth code seg is loaded.


; C Base program name.
;
; This is a counted ASCIIZ sting, with the count including the
; terminating zero
;
; If the count is 0 then no C Base program is loaded and control is
; passed directly to the Forth++ Code segment
;
; If the count is -1 then the ASCIIZ string holds the Complete path name
; Otherwise the file name is added to the end of the default load path
;

cname   db      0
        db      7fh DUP(?)


; The following equates give addresses in the Forth code segment which
; are used to hold the addresses of other segments.

fsptr   equ     06h      ; Forth segment pointer
nsptr   equ     08h      ; Names
$sptr   equ     0ah      ; Strings
ssptr   equ     0ch      ; Stack
lsptr   equ     0eh      ; Loader segment pointer

; Address for Forth to call to re-enter C system.  This is initialised
; to point to an error handler by SAVE-SYS and is reset by the C overlay
; initialisation.

roptr   equ     10h      ; Offset
rsptr   equ     12h      ; Segment

; Forth++ Segment file name extensions

cfile   db      "CDE",0   ; Code
ffile   db      "FOR",0   ; Forth
nfile   db      "NAM",0   ; Name
$file   db      "STR",0   ; String


; ===================================================================
;                           Loader  Program
; ===================================================================
;

; Initialise all segments to point to the same (code) segment

start:  cld
        mov     ax,cs
        mov     ds,ax
        mov     es,ax
```

```
        mov     ss,ax
        mov     sp,OFFSET stack

; As we use features of DOS that only appeared (documented) in DOS 3.0
; we must make sure that the user is not running an earlier DOS system.

        mov     bp,OFFSET msg_1         ; Point to the error message
        mov     ah,30h
        int     21h
        cmp     al,3
        jae     resize
        call    abort

; Resize memory back down so that we can grab it

resize: mov     bx,64h                  ; keep enough memory to work in
        mov     ah,4ah
        int     21h                     ; resize allocated memory
        jnc     cont
        call    mem_err

; Find the current program name.  This is stored at the end of the
; environment table.  The environment table can be a maximum of 32K.
; Each entry is terminated with a 00 byte and the table is terminated
; with a second 00 byte (after an entry terminating 00 byte).  The name
; of the currently executing program is then stored as an ASCIIZ string
; 2 bytes on from the end of the environment table.

cont:   mov     bp,OFFSET msg_3

        mov     ax,cs:2ch       ; Get segment addr of Env. table
        mov     es,ax
        mov     cx,8000h        ; Max size of Env. table
        xor     ax,ax
        mov     di,ax           ; Set DI to start of Env. table

scan:   or      cx,cx
        jnz     scn1
        call    abort           ; If table too big, Env. error
scn1:   repnz   scasb           ; Find end of entry
        scasb                   ; Is it end of table
        jnz     scan

        add     di,2            ; Move DI to start of program name

; The currently executing program name is the full path name of this
; program (eg, C:\FORTH\CFORTH\CFORTH.COM).  We copy this name into an
; internal buffer so that we can change it as required.
; This name can be no longer than 7F bytes.

        push    ds

        push    es
        mov     ax,ds
        mov     es,ax
        pop     ds

        mov     si,di
        mov     di,OFFSET fname

        mov     cx,7fh
        rep     movsb

        pop     ds
```

```
; Scan through the currently executing program name (in the buffer) to
; find the . used in the '.COM' at the end of the name.  This is so that
; we can simply replace the 'COM' part of the filename with the relevant
; extension required for a given overlay.
;
; It is possible to have a '.' in a directory name, hence we must find
; the end of the filename (a maximum of 7fh characters) and scan back
; towards the start to find the correct '.'.  The location of the
; character after the '.' is stored in the variable  fdot  for later use
; by the  loadseg  subroutine.

        mov     cl,80h          ; Max for file name + 1
        mov     al,0            ; Terminating character
        mov     di,OFFSET fname
        repnz   scasb
        je      sdot
        call    abort           ; Filename to long (Env. error)

sdot:   mov     cl,6            ; Max characters back + 1
        mov     al,'.'
        std
        repnz   scasb
        cld
        je      gdot
        call    abort           ; Can't find the '.' (Env. error)

gdot:   add     di,2
        mov     WORD PTR fdot,di


; Load the Forth++ system.

; The Forth++ system is made up of a set of overlay files.  The
; following code simply loads in each of the overlays in turn.  The
; overlay name is made up by taking the currently executing path name
; and replacing the .COM with the relevant extension for the given
; overlay.  The subroutine  loadseg  preforms most of the work required
; for this function.

        ; Code segment

        ; The code overlay is the first to be loaded as this has to be
        ; patched with the segment addresses of the remaining overlays.
        ; The segment address of the code overlay is placed in ES and
        ; the offset to patch is placed in BP.  When we load the Code
        ; overlay we place the address of  eseg  in ES:BP so we can
        ; patch in the segment required for the inter-segment jump into
        ; the Forth system.

        mov     bx,WORD PTR csize       ; Size of Segment
        mov     bp,OFFSET eseg          ; Offset in ES patch seg addr.
        mov     dx,OFFSET cfile         ; Segment extension
        call    loadseg                 ; Load the Forth segment

        mov     ax,WORD PTR eseg        ; Recover overlay segment addr
        mov     es,ax                   ; Set as ES

        ; Set  rsptr  field of Forth code segment to contain the segment
        ; address of the code overlay.  If a C overlay file is loaded
        ; this setting will be overwritten by the C code to the segment
        ; address of the C code.  This is to assist Forth to set up an
        ; error trap for spurious C calls (ie, those made when no C
        ; overlay is present).

        mov     es:[rsptr],ax
```

```
        ; Forth segment

        mov     bx,WORD PTR fsize
        mov     bp,fsptr
        mov     dx,OFFSET ffile
        call    loadseg

        ; Name segment

        mov     bx,WORD PTR nsize
        mov     bp,nsptr
        mov     dx,OFFSET nfile
        call    loadseg

        ; String segment

        mov     bx,WORD PTR $size
        mov     bp,$sptr
        mov     dx,OFFSET $file
        call    loadseg

        ; Stack segment

        ; Because the stack segment does not require to be initialised
        ; we do not have an overlay for it.  Here we simply obtain the
        ; memory required for the stack and patch the code overlay
        ; directly.

        mov     bx,WORD PTR ssize
        mov     ah,48h
        int     21h
        jnc     ssseg
        call    mem_err
ssseg:  mov     es:[ssptr],ax

        ; Load segment

        ; The  lsptr  field of the code overlay is set to the segment
        ; address of the currently execution program.  The setup chain
        ; of the Forth system will then be able to inspect (and execute)
        ; any text given on the command line after the program name.

        mov     ax,cs
        mov     es:[lsptr],ax


; Do we need to load in the C base program?
;
; The byte at  cname  is the count for an ASCIIZ string holding the name
; of the C overlay file.  If this count is 0 (zero) then the C overlay
; is not required, so execution is passed directly to the Forth Code
; overlay.

        mov     ax,ds
        mov     es,ax

        mov     si,OFFSET cname         ; Count byte of C overlay name
        mov     al,[si]
        inc     si
        cmp     al,0                    ; Is the count zero ?
        jne     covl                    ; No => Load the C overlay
        jmp     DWORD PTR eoff          ; Yes => jump to Forth code seg

; The byte at  cname  did not contain a 0 (zero), hence we must load
```

```
; (and execute) the C overlay file.  If the count byte is -1 the
; following 7fh bytes give the full pathname of the C overlay file.
; However, if the count is not -1, it should be the length of the
; filename (including the terminating zero) for the C overlay.  This
; filename will be appended to the current load path, used to load in
; the other overlays.

covl:   cmp     al,-1
        je      covl_2

        ; The count byte indicates the number of characters to add to
        ; the end of the current load pathname.  In order to do this we
        ; must first find the first character of the file name at the
        ; end of the most recently loaded overlay (string).  This can be
        ; done by scanning backwards from the . used to indicate the
        ; overlay type, looking for the \ used to indicate a directory
        ; name.  If the \ is not found within the maximum number of
        ; characters allowed for a file name (8 + 3) then an Environment
        ; error is indicated.

        mov     bp,OFFSET msg_3
        push    ax

        ; Find the '\'
        mov     cx,11
        mov     di,WORD PTR fdot
        mov     al,'\'
        std
        repne   scasb
        cld
        je      covl_1
        call    abort

covl_1: add     di,2

        ; Copy the given filename onto the end of the path
        pop     cx
        mov     ch,0
        rep     movsb

        ; Load and execute the overlay
        jmp     cload

        ; The count byte was -1, so copy the full pathname into the
        ; internal buffer.

covl_2: mov     di,OFFSET fname
        mov     cx,7fh
        rep     movsb

cload:  ; Load and execute the C overlay file.  The name of the file to
        ; load is in the internal name buffer  fname  .

        ; Initialise all seven loadblock fields to 0000h

        mov     di,OFFSET loadblock
        push    di
        mov     cl,7
        xor     ax,ax
        rep     stosw
        pop     di

        ; Initialise the new loadblock so that the C overlay inherits
        ; the eseg:eoff values as the first four bytes of its default
        ; FCB.  This is to pass the segment:offset address of the Forth
```

```
        ; code overlays entry address to the C system.

        mov     ax,cs
        mov     [di+8],ax       ; Default FCB segment
        mov     [di+4],ax       ; Command line
        mov     ax,OFFSET eoff
        mov     [di+6],ax       ; Default FCB holding eseg:eoff
        mov     ax,80h
        mov     [di+2],ax       ; Command line tail

        ; Invoke the program (C Overlay file).  Note the execution is
        ; passed to the C overlay file to allow the C initialisation
        ; code to be executed before the Forth system is invoked.  The
        ; overlay file will invoke the Forth system via the execution
        ; address passed to it in the default FCB.

        mov     dx,OFFSET fname
        mov     bx,di
        mov     ax,4b00h        ; Load and execute an overlay
        int     21h
        mov     bx,cs
        mov     ss,bx
        mov     sp,OFFSET stack
        jnc     execok
        call    load_err

execok: mov     al,0
        jmp     exit

;===========
; Subroutine:  Loadseg -> Load a given Forth++ Segment overlay
;===========
;
; On entry:
;          bx => No of paragraphs required for overlay
;       es:bp => addr to place segment addr of new overlay
;          dx => overlay name
;        fdot => addr of first char after the '.' in load file name

loadseg LABEL   NEAR
        push    es

        ; Grab the memory
        mov     ah,48h
        int     21h
        jnc     rdseg
        call    mem_err

        ; We got it and its seg addr is in ax - patch it into the
        ; code overlay.
rdseg:  mov     es:[bp],ax

        ; set es:bx to the parameter block "loadblock"
        push    cs
        pop     es
        mov     bx,OFFSET loadblock

        ; Initialise the load parameter block.
        mov     [bx],ax            ; destination segment
        mov     WORD PTR 2[bx],0   ; load Offset is set to zero

        ; Copy the overlay extension name over the .COM
        mov     si,dx
        mov     di,WORD PTR fdot
        mov     cl,3
```

```
        rep     movsb

        ; set ds:dx to file name

        mov     dx,OFFSET fname

        ; load file as overlay
        mov     ax,4b03h
        int     21h
        jnc     rdok
        call    load_err

rdok:   pop     es
        ret


;===========
; Subroutine: Mem_err -> Display a memory error condition and abort
;===========
;

mem_err:mov     bp,OFFSET msg_2

;===========
; Subroutine: Abort -> Display an error condition and abort
;===========
;
; BP => Error message to display
; AX => Error code

abort:  push    bp
        push    ax

        mov     bp,OFFSET msg_A  ; <CR><LF>Forth++ Load error <
        call    display

        pop     dx
        pop     bp
        pop     ax
        sub     ax,3
        call    hexw             ; Address of error
        mov     al,'/'
        call    char             ; Separator
        mov     ax,dx
        call    hexw             ; Error code (AX at time of error)

        call    display          ; Error text (for user) '> xxx'
        mov     bp,OFFSET msg_B  ;  !<CR><LF><BELL>
        call    display

        mov     al,1

;===========
; Subroutine: Exit - Exit back to Dos
;===========

exit:   mov     ah,4ch           ; Exit back to Dos
        int     21h
        jmp     exit

;===========
; Subroutine: Load_err -> Display a load error condition and abort
;===========
;
; fseg:foff => Address of ASCIIZ string holding file name
```

```
load_err:
        mov     dx,ax

        mov     bp,OFFSET msg_A  ; <CR><LF>Forth++ Load error <
        call    display

        pop     ax
        sub     ax,3
        call    hexw             ; Execution address
        mov     al,':'           ; Separator char
        call    char
        mov     ax,dx            ; Error code
        call    hexw

        mov     bp,OFFSET msg_5  ; > Can't find:
        cmp     dx,2
        je      le_1
        cmp     dx,3
        je      le_1

        mov     bp,OFFSET msg_6  ; > access denied when loading <CR><LF>
        cmp     dx,5
        je      le_1

        mov     bp,OFFSET msg_2  ; > Out of memory
        cmp     dx,8
        je      le_2

        mov     bp,OFFSET msg_3  ; > Environment error
        cmp     dx,0ah
        je      le_2

        mov     bp,OFFSET msg_4  ; > in

le_1:   call    display          ; Display error message

        mov     bp,OFFSET fname  ; Display File name (or err msg)
le_2:   call    display

        mov     bp,OFFSET msg_B  ;  !<CR><LF><BELL>
        call    display
        mov     al,1
        jmp     exit

;===========
; Subroutine: Display -> Display a ASCIIZ string
;===========
;
; BP => addr of ASCIIZ string to be displayed
;

display:cld
        push    si
        mov     si,bp
d1:     lodsb
        cmp     al,0
        je      dr
        call    char
        jmp     d1
dr:     pop     si
        ret

;===========
; Subroutine: HexW -> Display a hex word
```

```
;===========
;
; AX => Word to be displayed as four hex digits
;

hexw:   push    ax
        mov     al,ah
        call    hexb
        pop     ax

;===========
; Subroutine: HexB -> Display a hex byte
;===========
;
; AL => Byte to be displayed as two hex digits
;

hexb:   push    ax
        mov     cl,4            ;***  This is required as MASM (4.0) is
        shr     al,cl           ;***  not able to accept: shr al,4
        call    hexc
        pop     ax

;===========
; Subroutine: HexC -> Display a hex character
;===========
;
; AL => Bits 0..3 to be displayed as a single hex digit
;

hexc:   and     al,0fh
        add     al,'0'
        cmp     al,'9'+1
        jc      char
        add     al,'A'-('9'+1)

;===========
; Subroutine: Char -> Display a character on the video
;===========
;
; On entry:
;       AL => Character to be displayed
;

char:   push    dx
        mov     dl,al
        mov     ah,2
        int     21h
        pop     dx
        ret

;=========================================================================
;                               DATA AREA
;=========================================================================

; Error messages

msg_A:  db      13,10,'Forth++ Load error <',0
msg_B:  db      ' !',13,10,7,0

; The DOS must be version 3.0 or above

msg_1:  db      '> Dos 3.0 (or above) required',0

; An out of memory error occurs when a request for memory is denied.
```

```
msg_2:  db      '> Out of Memory',0

; An Environment error occurs when:
; 1. The environment table is longer than its maximum 32K.
; 2. The currently executing file name is longer than its max (7F bytes)
; 3. The . can not be found at the end of the file name
; 4. The \ can not be found at the start of the file name
; 5. Returned as an error from the load (4B) function

msg_3:  db      '> Environment Error',0

; When the load (or load and execute) overlay function (4B) is invoked,
; it may return an error code.  The following error messages will be
; displayed dependent on the error code returned.
;
;     Out of memory - Insufficient memory to load the overlay
; Environment error - Bad Environment
;        Can't find - File does not exist
;     Access denied - File access is denied
;                in - Any other error condition

msg_4:  db      '> in ',0

msg_5:  db      '> Can''t find: ',0

msg_6:  db      '> Access denied when loading',13,10,0


;=========================================================================
;                           Variable Space
;=========================================================================

; The offset address of the name that is attempting to be loaded (in
; case of an Can't find error) is stored in an internal variable.

foff    dw        ?


; The loadblock used to load the overlays (and C base program)

loadblock dw      7 DUP(?)


; A buffer to store the currently executing program name.  The name in
; this buffer will be manipulated to form the correct path name for the
; overlay files that make up the CForth++ system.

fdot    dw        ?
fname   db        80h DUP(?)

; Some stack area.

stack   dw        160 DUP(?)

entry   ENDP
CFL_TEXT ends
        end     entry
```

In order to generate the loader program, you should invoke the `MAKELOAD` batch file provided. This will produce the (`.COM` format) loader template `FLOADER.TPT`. This is a simple data file, loaded into memory (from 0100h) of the current load segment.

As this program is less than 2 Kbytes in length, it means that we can store it as a couple of FORTH blocks directly. When we need to alter the variables we can simply copy the FORTH blocks to the required file name to produce the new program loader.

C.2 Making the loader

The `MAKELOAD` batch file will process the `CFLOAD.ASM` file into the system loader (`FLOADER.TPT`). The file is assembled, linked and then converted into the '`.COM`' format before being renamed `FLOADER.TPT` as required by FORTH++'s `SAVE-SYS` command.

```
rem *** Assemble the loader ***
masm cfload , , , ,

rem *** Link it into a .EXE file ***
link cfload ;

rem *** Now convert it into a .BIN ***
exe2bin cfload.exe

rem *** Copy it to the FLOADER.TPT template ***
copy cfload.bin floader.tpt

rem *** Delete unwanted files ***
del CFLOAD.LST
del CFLOAD.OBJ
del CFLOAD.CRF
del CFLOAD.EXE
del CFLOAD.BIN
```

C.3 Overlay initialisation

This is the initial C code that is executed immediately after the loader program has loaded in the C module. It simply calls the assembler code routine `FINIT` to initialise the FORTH interface and then enters into a tight loop passing control to the FORTH system. When control is returned to the C system, it pops an integer value off from the FORTH stack and uses it as an index into a jump table of C routines. The C routine is then executed and control is passed back to the FORTH system.

```
#include "cforth.h"

/*************************************************************************/
/***                                                                   ***/
/***                    CFORTH1.C Definitions                          ***/
/***                                                                   ***/
/***   The following initialised structures are defined by the         ***/
/***   customer code, in the file CFORTH1.C                            ***/
/***                                                                   ***/
/***                                                                   ***/

/*
 * jmptbl[] is a table that contains functions that can be invoked from
 * the Forth system.  A function number is used as an index into the
 * table.
 */

extern TBL jmptbl[];

/*
 * The function startup() is invoked after initialisation of the C
```

```
 * system to allow the customer code to perform any initialisation that
 * it may require.
 */

extern void startup(void);

/***                                                                  ***/
/***                  End of CFORTH.C definitions                     ***/
/***                                                                  ***/
/**********************************************************************/

/**********************************************************************/
/***                                                                  ***/
/***                        Forth Interface                           ***/
/***                                                                  ***/
/***  The following code is for the C Main function.  This code is    ***/
/***  the interface between C system and the Forth system.  It        ***/
/***  also provides the calling mechanism to allow the Forth system   ***/
/***  to invoke the C functions given in the jump table.  The         ***/
/***  external functions (FINIT and FORTH) can be found in the MASM   ***/
/***  assembly source code file CFASM.ASM.                            ***/
/***                                                                  ***/
/***  This code should NOT be altered unless you are sure about it!   ***/
/***                                                                  ***/
/**********************************************************************/

/*
 * Declare the external values.  The function FINIT() is called to
 * initialise the Forth++ system.  It will initialise the context
 * switching area.  The function FORTH() will save the C environment,
 * build the Forth environment and then continue execution of the Forth
 * system.  When the Forth system wants to invoke a C function it will
 * return to FORTH().  FORTH() will then swap execution environments and
 * return execution to the C system.  The Forth Stack Pointer is updated
 * on entry/exit of the Forth system.
 */

/*
 * Under Zortech C++ we must declare these functions as having C type
 * linking.  Hence the next two lines would be:
 *
 *              extern "C" { void FINIT(void); }
 *              extern "C" { void FORTH(void); }
 *
 * Note: This is only required for C++, the Zortech C compiler will
 *       error when given this code.
 */

extern void far FINIT(void);
extern void far FORTH(void);

main(void)
{
 /* Invoke the Forth initialisation code */

 FINIT();

 /* Invoke customer C start up code */

 startup();

 /* Execute Forth and interpret any C calls */

 {
  unsigned int i;
```

```
  while(1)
    {
     FORTH();
     i = POP(int);
     jmptbl[i].function();
    }
 }
}
```

C.4 Context Switching

This is the code that actually does the hard work of transferring control between the FORTH and C systems. This file is designed as a C module and is to be linked in with the users C code.

There are two assembler code routines in this file. **_FINIT** is called by the C initialisation code to initialise the C to FORTH data required by the **_FORTH** code. This will set up the initial FORTH entry address (including Code segment) as passed to it from the loader program (via the program segment prefix).

The **_FORTH** subroutine is called by the C when it wants to transfer control from the C system to the FORTH system. This subroutine simply stores the state of the C system (on the C stack) and then recovers the state of the FORTH system[1]. It will then pass control to the FORTH system.

When the FORTH system wants to pass control back to the C system, it will make an Inter-segment call to the label **freturn**. This code will save the current FORTH status, recover the C status (from the C stack) and return to the calling C code.

```
        PAGE    66,132
        TITLE   C to FORTH Interface  (P.J. Knaggs 08/08/90)

; This code is included as part of the C overlay.  It contains the MASM
; assembler code for the actual C to Forth interface.  Two routines are
; provided to be linked with the C system, they are _FINIT and _FORTH.

CFASM_TEXT segment byte public 'CODE'
        assume  cs:CFASM_TEXT,ds:_DATA

        SUBTTL  Initialise the Forth++ System

        PUBLIC  _FINIT
_FINIT  PROC    far

; void far _FINIT(void)
;
; Initialise the C data area for the context switching.  Also initialise
; the remaining part of the Forth system for the C overlay.

; The following equates give address in the Forth Code segment
; used to hold the address of the code to invoke the C

roptr   equ     10h     ; Return Offset pointer
rsptr   equ     12h     ; Return Segment pointer

ssptr   equ     0ch     ; Stack Segment pointer

; Save the C environment (on the C stack)

        pushf
        push    si
        push    di
```

[1] From variables, as the C system needs access to the FORTH stack structure for argument handling

```
        push    bp
        push    es
        push    ds

        mov     ax,_DATA
        mov     ds,ax

; Set Direction Flag to increment

        cld

; Extract the execution seg & offset of the Forth system from locations
; 5ch and 5eh of the PSP, where they will have been deposited by the
; Forth system loader.  Locations 5c and 5e are safe to use as they are
; in redundant PSP locations (actually a file control block).

        mov     ah,62h
        int     21h
        mov     es,bx

        ; Store the execution seg & offset in eseg & eoff

        mov     ax,WORD PTR es:5ch      ; Execution offset
        mov     eoff,ax

        mov     ax,WORD PTR es:5eh      ; Execution (Code) Segment
        mov     eseg,ax
        mov     es,ax

; We can now place the C execution vector into the Forth code segment.
; This is the address of the code that the Forth system is to execute
; when it transfers control back to the C system.  This is set to an
; error reporter by SAVE-SYS, but we now replace it to point to freturn.

        mov     ax,SEG freturn
        mov     es:[rsptr],ax           ; Return Seg Pointer
        mov     ax,OFFSET freturn
        mov     es:[roptr],ax           ; Return Offset Ptr

; We must set up the Forth stack (segment and offset) so that on the
; first execution of _FORTH the stack is at a sensible location.
; Thereafter it will be looked after by the Forth system.

        mov     ax,es:[ssptr]
        mov     WORD PTR sseg,ax
        mov     ax,4
        mov     WORD PTR soff,ax

; Recover the C environment from the stack

        pop     ds
        pop     es
        pop     bp
        pop     di
        pop     si
        popf
        ret
_FINIT  ENDP


        SUBTTL  Switch context between C and Forth and back again

        PUBLIC  _FORTH
_FORTH  PROC    far
```

```
; void FORTH(void) - Enter into the forth system
;
; When the C system has completed its task it will transfer control
; to the Forth system by executing this code.  When the Forth system
; wants to re-enter the C system it will invoke the code at freturn.
;
; The C environment is stored on the C stack before the SS:SP address
; is stored in cseg:coff.  All registers not saved can be discarded or
; recovered by the program.

; Save the C environment

        pushf
        push    si
        push    di
        push    bp
        push    es
        push    ds

; Save the C stack

        mov     ax,_DATA
        mov     ds,ax
        mov     cseg,ss
        mov     coff,sp

; Read Forth stack (from __FSP)

        mov     sp,soff
        mov     ss,sseg

; Set up for re-entry to Forth with a far return

        push    eseg
        push    eoff

; Restore Forth environment

        ; Forth++ uses the following registers:
        ;       SI, BP, BX, CS, DS, SS and SP
        ;
        ; We are changing the stack pointer so we must generate
        ; the new values of SS:SP (stored in the variable __FSP)
        ;
        ; The CS:IP value will be set when we return to the forth
        ; system.  The values are stored in eseg:eoff
        ;
        ; The remaining registers are stored in an environment buffer.
        ; They can not be stored on the Forth stack as the C code
        ; requires access to the Forth stack for argument passing, thus
        ; these values are stored in this environment buffer  fbuf .

        mov     si,fiip
        mov     bp,frsp
        mov     bx,fubp
        mov     ds,fds

; Clear direction flag (required in Forth inner interpreter)

        cld

; Execute the Forth system

        ret
```

```
; Forth to C
;
; When Forth is ready to invoke a C function, it will make an inter-
; segment call to the following code.  This will return to the C code
; which will, in turn, pop an integer value from the Forth stack and
; execute the corresponding jump table entry.

freturn label   near

; Set the data segment to access interface data

        push    ds
        mov     ax,_DATA
        mov     ds,ax

; Save Forth's environment

        ; As the C system requires access to the Forth stack (to get the
        ; function request number and any other argument passing) we
        ; can't store the systems state on the Forth stack.  Thus we
        ; must store it in the "environment buffer",  fbuf   .

        mov     fiip,si
        mov     frsp,bp
        mov     fubp,bx
        pop     fds

        ; Save Forth's re-entry point

        pop     eoff
        pop     eseg

        ; Save Forth's stack (in __FSP)

        mov     sseg,ss
        mov     soff,sp

; Recover the C environment

        ; Recover the C stack

        mov     ss,cseg
        mov     sp,coff

        ; Recover the C registers

        pop     ds
        pop     es
        pop     bp
        pop     di
        pop     si
        popf

; Return to the C system
        ret
_FORTH  ENDP

CFASM_TEXT ends

        SUBTTL  Data Area

_DATA   segment word public 'DATA'

; This is the Data Area required by the _FORTH function.
```

```
; The Following locations are the store for the Forth++ Environment

fbuf    label   word
fiip    dw      ?       ; Inner Interpreter ptr
frsp    dw      ?       ; Return Stack ptr
fubp    dw      ?       ; User Base ptr
fds     dw      ?       ; Data Segment

; Space to hold the C stack pointer

coff    dw      ?
cseg    dw      ?

; Space to hold the Forth Stack Pointer.

        PUBLIC  __FSP           ; Used by C to manipulate Forth Stack
__FSP   label   dword
soff    dw      ?
sseg    dw      ?

; Space to hold the Execution Address of the Forth system

eoff    dw      ?
eseg    dw      ?

_DATA   ends

        end
```

This module is merged with the `CFINIT.C` module to form a library. This makes the linking process much simpler. In order to build this library one must first obtain the "object code" for the two modules. To assemble the `CFASM.ASM` file one would give the command:

```
MASM CFASM, CFASM, NUL, NUL /mx
```

While the `CFINIT.C` modules needs to be compiled:

```
TCC -c -ml -G CFINIT
```

Here we are compiling the C module with Borland's "Turbo C" compiler. The '`-c`' indicates that we want to compile the file. The '`-G`' instructs the compiler to optimise the code for speed rather than size. Finally the '`-ml`' option instructs the compiler to use the "*L*arge" memory model.

We are now in a position to make the library. This we do by giving the command:

```
TLIB CFORTH /C +CFASM +CFINIT
```

In this instruction we are asking the system to generate a library named `CFORTH.LIB` which is case sensitive with regard to the public labels. This library consists of merging the two object code files `CFASM.OBJ` and `CFINIT.OBJ` that we have just produced.

C.5 Stack access

C access to the FORTH stack is provided via a set of type independent macros. The header file `CFORTH.H` defines these macros, it should be included into the users C code.

```
/*****************************************************************/
/***                                                           ***/
/***                        CFORTH.H                           ***/
/***                                                           ***/
```

```
/***   The following lines of code make up the file CFORTH.H.    ***/
/***                                                             ***/

/*
 * entry() is a macro definition designed to help in the setting up of
 * the jump table.  To use this macro you must be defining the jump
 * table, simply type:
 *
 *               entry(func)
 *
 * where func is the name of the code required for the given entry.
 */

#define entry(func)     { func }

/***                                                             ***
 *** The following definitions set up the types required by the  ***
 *** jump table.                                                 ***
 ***                                                             ***/

/*
 * The type PFI is defined to be a Pointer to an Function returning an
 * Integer.
 */

typedef int (*PFI)();

/*
 * The table entry structure is defined to be of type PFI.
 */

#define TBL     struct tabentry
TBL { PFI function; };

/*****************************************************************/
/*                                                               */
/*                      Stack Manipulations                      */
/*                                                               */
/* The following functions are defined to Manipulate the Forth  */
/* systems stack.  Items can be popped off the stack and pushed */
/* onto it.  All communication between the C system and the      */
/* Forth system should be conducted via these functions.        */
/*                                                               */
/*****************************************************************/

/*
 * The Forth Stack Pointer is stored as a far pointer to a void.
 * This is stored in the CFASM module for access by the assembly
 * code.
 */

extern void far * _FSP;

/*
 * The DROP(type) macro is defined to drop an item of the given type
 * from the Forth stack.
 */

#define DROP(type)      (type far *)_FSP += 1

/*
 * The macro INDEX(type,n) is used to return the n-th stack item of the
 * given type.  Hence INDEX(int,0) will return the TOP int on the stack,
 * while INDEX(double,1) will return the second double on the stack.
 */
```

```
#define INDEX(type,n)   *( (type far *)_FSP+n )

/*
 * The macro IINDEX(type,offset) is used in the same manner as the
 * INDEX() macro except that the offset value is in stack cells and has
 * no regard to type size.
 */

#define IINDEX(type,n)  *( (type far *) ( (int far *)_FSP+n ) )

/*
 * The macro POP(type) is used to POP an item of the given type from the
 * Forth stack.
 */

#define POP(type)       *(type far *)_FSP; (type far *)_FSP += 1

/*
 * The PUSH(type,val) macro is used to PUSH a given value (val) of the
 * given type onto the Forth stack.
 */

#define PUSH(type,n)    (type far *)_FSP -= 1; *(type far *)_FSP = n

/*
 * Note:
 * The macros POP and PUSH should have been defined as follows:
 *
 *      #define POP(type)       *(type far *)(_FSP++);
 *      #define PUSH(type,n)    *(type far *)(--_FSP) = n;
 *
 * However Zortech C was the only system that could handle this complex
 * a definition.  For both Microsoft and Borland the definition has to
 * be split as above.
 */

/***                                                                  ***/
/***                     End of CFORTH.H                              ***/
/***                                                                  ***/
/************************************************************************/
```

C.6 User code

The user places his code in a separate "user" module. The file `CFORTH1.C` is an example user module. In this example module, we provide a floating point maths system using the C floating point code rather than developing our own.

The user *must* provide the '`jmptbl`' jump table. The routine '`setup`' must also be provided to initialise any user code.

The user can write any C code they like in this file. The functions that they want to be accessible from FORTH must be given in the jump table.

A function that is to be invoked from FORTH must not have any arguments and must not return a value. All argument passing should be performed by means of the macros provided in `CFORTH.H`. These macros actually manipulate the FORTH stack directly (as a C data structure).

```
/************************************************************************/
/***                                                                  ***/
/***                          CFORTH Header                           ***/
/***                                                                  ***/
/***  You must include the CFORTH.H header file in order to define  ***/
```

```
/***  the macros used to manipulate the Forth stack.  It also       ***/
/***  defines the structure of the jump table.                      ***/
/***                                                                ***/
/********************************************************************/

#include "cforth.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/********************************************************************/
/***                                                              ***/
/***                     User Functions:                          ***/
/***                                                              ***/
/***  The following are the C functions which are to be called by   ***/
/***  Forth.  After they have been defined the functions will be    ***/
/***  placed in a jump table.  These functions do not pass          ***/
/***  parameters in normal C fashion and must have the prototype:   ***/
/***                                                              ***/
/***     int func(void)                                           ***/
/***                                                              ***/
/***  Ie, default functions taking no arguments.  Actually the      ***/
/***  functions pass their arguments via the Forth stack, which     ***/
/***  appears to the C program as a data structure.  Forth stack    ***/
/***  arguments are accessed with PUSH and POP macros.              ***/
/***                                                              ***/
/***  To pop a value use:      x = POP(type);  eg, x = POP(int);    ***/
/***  To push a value use:     PUSH(type,x);   eg, PUSH(int,-1);    ***/
/***                                                              ***/
/********************************************************************/

/*
 * Declare variables to hold forth segment and offset.  A Forth call
 * will set fseg to the Forth memory space segment address. fseg and
 * ofs are used when a 16 bit Forth memory address is passed to a C
 * function.
 */

unsigned fseg, ofs;

/***                                                          ***
 ***            Provide floating point support.               ***
 ***                                                          ***/

/*
 * declare floating point stack and stack pointer.
 */

double fs[100], *fsp;
int base_pointer;

/***                                                              ***
 ***  Set up initial values for floating point stack pointer and    ***
 ***  floating point base pointer.  A margin of 5 elements is       ***
 ***  allowed to cater for stack underflow, which Forth will check  ***
 ***  for after interpreting each command line.                     ***
 ***                                                              ***/

call_finit() { fsp=fs+5; base_pointer = 5; }

/***                                                              ***
 ***  The following code is invoked by the Forth system to set the  ***
 ***  fseg variable to the Forth data segment.  Thus allowing far   ***
 ***  addresses to be calculated (with MK_FP).                      ***
 ***                                                              ***/
```

```
setfseg() { fseg = POP(int); }

/*********************************************************************
 ***                                                               ***
 ***                  Floating point functions                     ***
 ***                                                               ***
 *********************************************************************/

call_fdiv() { *(fsp-1) = *(fsp-1) / *fsp; fsp-- ; }

call_float() { ++fsp; *fsp = (double) POP(int); }

call_fminus() { *(fsp-1) = *(fsp-1) - *fsp; fsp-- ; }

call_fmult() { *(fsp-1) = *(fsp-1) * *fsp; fsp-- ; }

call_fplus() { *(fsp-1) = *(fsp-1) + *fsp; fsp-- ; }

call_int() {
  int n;
  n = (int) *fsp; fsp--; PUSH(int,n); }

fppfrom() { PUSH(float,*fsp); fsp--; }

tofpp() { ++fsp; *fsp = POP(float); }

fdup() { ++fsp; *fsp = *(fsp-1); }

fdrop() { --fsp; }

fswap() {
  double temp;
  temp = *fsp; *fsp = *(fsp-1); *(fsp-1) = temp; }

fover() { ++fsp; *fsp = *(fsp-2); }

frot() {
  double temp;
  temp = *fsp; *fsp = *(fsp-2); *(fsp-2) = *(fsp-1); *(fsp-1) = temp; }

setstackpointer() { fsp = fs + POP(int); }

fetchstackpointer() { PUSH(int,fsp-fs); }

setbasepointer() { base_pointer = POP(int); }

fetchbasepointer() { PUSH(int,base_pointer); }

framefetch() {
  int index;
  index = POP(int); ++fsp; *fsp = fs[index+base_pointer+1]; }

framestore() {
  int index;
  index = POP(int); fs[index+base_pointer+1] = *fsp; --fsp; }

d_to_f() { ++fsp; *fsp = POP(long); }

f_to_d() { PUSH(long,*fsp); --fsp; }

fless() {
  if (*(fsp-1) < *fsp) { PUSH(int,-1); } else { PUSH(int,0); }
   --fsp; --fsp; }
```

```
fgreater() {
  if (*(fsp-1) > *fsp) { PUSH(int,-1); } else { PUSH(int,0); }
   --fsp; --fsp; }

fequal() {
  if (*(fsp-1) == *fsp) { PUSH(int,-1); } else { PUSH(int,0); }
   --fsp; --fsp; }

f0greater() {
  if (*fsp > 0) { PUSH(int,-1); } else { PUSH(int,0); }
  --fsp; }

f0less() {
  if (*fsp < 0) { PUSH(int,-1); } else { PUSH(int,0); }
  --fsp; }

f0equal() {
   if (*fsp == 0) { PUSH(int,-1); } else { PUSH(int,0); }
  --fsp; }

fdepth() { PUSH(int, fsp-fs-base_pointer); }

call_acos() { *fsp = acos(*fsp); }

call_asin() { *fsp = asin(*fsp); }

call_atan2() { *(fsp-1) = atan2(*fsp,*(fsp-1)); --fsp; }

call_cos() { *fsp = cos(*fsp); }

call_sin() { *fsp = sin(*fsp); }

call_cosh() { *fsp = cosh(*fsp); }

call_sinh() { *fsp = sinh(*fsp); }

sincos() {
  double temp;
  temp = *fsp; *fsp++ = sin(temp); *fsp = cos(temp); }

call_exp() { *fsp = exp(*fsp); }

call_fabs() { *fsp = fabs(*fsp); }

call_floor() { *fsp = floor(*fsp); }

call_frexp() {
  int n;
  *fsp = frexp(*fsp,&n); PUSH(int,n); }

call_ldexp() {
  int exp;
  exp = POP(int); *fsp = ldexp(*fsp,exp); }

call_log() { *fsp = log(*fsp); }

call_log10() { *fsp = log10(*fsp); }

call_modf() { ++fsp; *fsp = modf(*fsp,fsp-1); }

call_pow() { *(fsp-1) = pow(*(fsp-1),*fsp); --fsp; }

call_pow10() { *fsp = pow10(*fsp); }

call_sqrt() { *fsp = sqrt(*fsp); }
```

```
fround() {
  double ipart;
  if( modf(*fsp,&ipart) >= 0.5 ) ipart=ipart+1;
  *fsp = ipart; }

fmax() { if( *(fsp-1) < *fsp) *(fsp-1) = *fsp; --fsp; }

fmin() { if( *(fsp-1) > *fsp) *(fsp-1) = *fsp; --fsp; }

fnegate() { *fsp = -*fsp; }

dfpp_from() { PUSH(double,*fsp); --fsp; }

to_dfpp() { ++fsp; *fsp = POP(double); }

call_atan() { *fsp = atan(*fsp); }

call_tan() { *fsp = tan(*fsp); }

dfp_store() {
  double far *ptr;
  ofs = POP(int); ptr = MK_FP(fseg,ofs); *ptr = *fsp; --fsp; }

dfp_fetch() {
  double far *ptr;
  ofs = POP(int); ++fsp; ptr = MK_FP(fseg,ofs); *fsp = *ptr; }

/*********************************************************************/
/***                                                               ***/
/***                          Jump Table                           ***/
/***                                                               ***/
/***  The function table follows.  Functions placed in this table  ***/
/***  are invoked by number when a C function call is received     ***/
/***  from Forth.  The code to process Forth C calls and invoke    ***/
/***  these functions is in CFINIT.C.  The entry macro, which in   ***/
/***  used to place entries into this table, is defined in the     ***/
/***  header file CFORTH.H                                         ***/
/***                                                               ***/
/*********************************************************************/


TBL jmptbl [] =
{
/* floating point initialisation */
entry(setfseg),

/* floating point support */
entry(call_fdiv),         entry(call_finit),    entry(call_float),
entry(call_fminus),       entry(call_fmult),    entry(call_fplus),
entry(call_int),          entry(fppfrom),       entry(tofpp),
entry(fdup),              entry(fdrop),         entry(fswap),
entry(fover),             entry(frot),          entry(setstackpointer),
entry(fetchstackpointer), entry(setbasepointer),entry(fetchbasepointer),
entry(framefetch),        entry(framestore),    entry(d_to_f),
entry(f_to_d),            entry(fless),         entry(fgreater),
entry(fequal),            entry(f0greater),     entry(f0less),
entry(f0equal),           entry(fdepth),        entry(call_acos),
entry(call_asin),         entry(call_atan2),    entry(call_cos),
entry(call_sin),          entry(call_cosh),     entry(call_sinh),
entry(sincos),            entry(call_exp),      entry(call_fabs),
entry(call_floor),        entry(call_frexp),    entry(call_ldexp),
entry(call_log),          entry(call_log10),    entry(call_modf),
entry(call_pow),          entry(call_pow10),    entry(call_sqrt),
entry(fround),            entry(fmax),          entry(fmin),
```

```
entry(fnegate),          entry(dfpp_from),     entry(to_dfpp),
entry(call_atan),        entry(call_tan),      entry(dfp_store),
entry(dfp_fetch)
};

/*************************************************************************/
/***                                                                   ***/
/***                         Start Up                                  ***/
/***                                                                   ***/
/***  The following code is executed as part of the initialisation  ***/
/***  sequence.  It should perform any initialisation required by   ***/
/***  the C code.                                                      ***/
/***                                                                   ***/
/*************************************************************************/

void startup()
{
 call_finit();
}
```

C.7 Making the C Overlay

To make the C overlay, one must first compile the user code `CFORTH1.C` with the command:

```
TCC -c -ml -G -d CFORTH1
```

The '`-d`' option is instructing the compiler to merge any duplicate strings it may find.

Having compiled the user code successfully we are able to link the file with the `CFORTH.LIB` library that we produced earlier:

```
TLINK /c C0L CFORTH1, CFORTH.OVL, CFORTH, CFORTH CL
```

In this command, we are linking the user code `CFORTH1.OBJ` with the standard prefix code `C0L.OBJ`, the library `CFORTH.LIB` and the standard C library `CL.LIB`. We have instructed the system to produce the overlay file `CFORTH1.OVL`. In addition to this, the command will also produce a map file `CFORTH.MAP`. Finally the '`/c`' option is given to instruct the system that all label names are case sensitive.

As MS-DOS does not provide a "make" facility, we provide two batch files that will compile the C/FORTH interface library and the users code into the C overlay that would be loaded by the CFORTH++ system. The `MAKEOVLL.BAT` file uses the *L*arge memory model:

```
rem *** Make the library ***
masm cfasm cfasm nul nul /mx
tcc -c -ml -G -d -r- cfinit
tlib cforth /C -+cfasm -+cfinit

rem *** Compile the overlay CFORTH1.C ***
tcc -c -ml -G -d cforth1

rem *** Link it with the Large libraries ***
tlink /x \tc\lib\c0l cforth1, cforth1.ovl, cforth, cforth \tc\lib\mathl
         \tc\lib\emu \tc\lib\graphics \tc\lib\cl
```

It is also possible to use the *S*mall memory model. For this one should use the `MAKEOVLS.BAT` batch file. This should be used if at all possible, as the C system requires less overhead for a small model, rather than the large memory model.

```
rem *** Make the library ***
masm cfasm cfasm nul nul /mx
```

```
tcc -c -ms -G -d -r- cfinit
tlib cforth /C -+cfasm -+cfinit

rem *** Compile the overlay CFORTH1.C ***
tcc -c -ms -G -d cforth1

rem *** Link with the Small libraries ***
tlink /x \tc\lib\c0s cforth1, cforth1.ovl, cforth, cforth \tc\lib\maths
         \tc\lib\emu \tc\lib\graphics \tc\lib\cs
```

This batch file differs from the previous only in that it compiles both the `CFINIT` module and the `CFORTH1` user module using the small memory model. It also links the system together with the standard small libraries (`c0s`, `maths` and `cs` as opposed to `c0l`, `mathl` and `cl`).