THInC - An Interactive and Extensible FORTH Compiler

Klaus D. Veil

Abstract

The FORTH programming approach has been acclaimed as an effective productivity tool mainly due to its interactiveness and extensibility. However, the advent of native-code compilers for the other languages has resulted in quite unfavourable FORTH runtime benchmark comparisons. The Threaded High-level Interactive nano-Compiler (THInC) system has been implemented as an attempt to alleviate this shortcoming while maintaining all the productive features of FORTH.

Introduction

To get a processor to solve computing tasks the gap between the abstract description of that task and the processor hardware must be "bridged" by a programming language and its compiler. The easier the use of the language and the better it utilises the processor's hardware, the more efficient the processor will solve the given computing task.

The FORTH language has made two significant contributions [1] to the ease of programming: Interactiveness allows the user to immediately see and check the results of the programmed command sequences. Extensibility allows the user unlimited extending of the available language, thus expanding it UP to match the programming task rather than deviding the problem DOWN to the level of the keywords. It is obvious that this approach will inherently lead to a richer language, more reusable code and higher productivity. Human communication would be very inefficient if limited to less than 100 keywords!

At the hardware end of this "bridge" compilers recently available for other languages (Pascal, C, etc.) generate quite fast and compact machine code. FORTH however has traditionally generated a pseudocode in the form of "indirect threaded code (ITC)" i.e. lists of addresses indirectly referencing the machine-code to execute the required function [2]. At run-time these address lists are then executed by a software "inner interpreter" [3]. While this allows any function to be called with a mere 2 bytes on a 8-bit processor, it does result in a substantial degradation of run-time speed.

To improve FORTH's run-time speed it has been suggested to replace ITC with "direct threaded code (DTC)" [4]. Some FORTH vendors have implemented "subroutine threaded code (STC)" [5] which generates lists of subroutine calls thus replacing the software inner inter-

preter with the processor's hardware JSR-RET mechanism. To achieve faster run-time behaviour, highly processor-dependant approaches have been taken [6].

It has also been suggested to use a second set of FORTH keywords to allow the programmer to generate in-line machine-code in time-critical parts of the program [7] [8]. It has even been suggested to do a straight macro expansion for each FORTH word [9] [10].

A quite different approach has been made possible with the advent of native FORTH chips. These provide the hardware to execute most of the basic FORTH keywords in one or two clockcycles without the intermediate steps of machine-code and micro-code [11]. This allows a processor throughput exceeding 8 million FORTH instructions per second! However, it is expected that the majority of systems will still have to utilise the more conventional processors.

The THInC Concept

The THInC approach to generating machine-code is essentially different: Each FORTH keyword is in essence a small but highly specialised compiler ("nano-Compiler") that produces the optimal machine code to effect the function of that particular keyword. The invocation of a keyword/compiler could compile some in-line code, a subroutine call or no code at all - possibly effecting a change of the compiler status only. The system consists of a multitude of these "nano-Compilers" which each handle the code generation for one traditional FORTH keyword. In true FORTH style this allows the full power of the language to be used for code generation while still maintaining enough system simplicity to enable the user to understand, maintain and extend the compiler.

The structure of each nano-Compiler, which is identical for most of the keywords, is inherently generalised and processorindependant. This allows further nano-Compilers to be added to the language just like FORTH allows additional commands to be defined. This results in an open and extensible FORTH compiler. This also allows an effortless implementation of THInC for any processor.

Most importantly, the THInC system is not a "post-processor" or "code optimiser" but is a fully live interactive and extensible FORTH programming environment.

The generated code can be for the host processor ("programming environment") or for a different processor ("cross-compiler").

Implementation

The THInC system is based on the concept of a two-stack machine; each stack controlling one of the two basic domains of a processor: data and instructions. It is interesting to note that about the same time as efforts were made to put the first real two-stack three-bus processor into silicon this concept surfaced during work on very early versions of THInC [12]. If possible, the top values of both stacks are held in processor registers. This results in significant speed improvements.

When a keyword is invoked, its "nano-Compiler" actively generates code to effect the action of that word. This is fundamentally different to FORTH were the machine-code belonging to a keyword is executed or an indirect pointer to it is compiled. The code compiled by THInC depends on the context the keyword is used in. The addition function "+" is a good example of nano-Compiler "intelligence":

If it is requested to add two values assumed on the stack, it could compile something like "(SP) Acc ADD SP INC SP INC" to add the top of the data-stack (held in Acc) to the second value on the data-stack (pointed to by SP) and then adjusting SP.

If the nano-Compiler detects that the top value is a literal (i.e. "1234 +") it could compile "1234 Acc ADI" to immediately add 1234 to the Accumulator. However, if this literal was found to be "1" it might be more efficient to compile: "Acc INC"! If the literal was 0, which could happen if it is hidden in a constant changed maybe for different software versions, no code would be generated at all. Similarly, a multiplication by 1 would produce no code, while "2 *" would compile "Acc SHL". This "strength reduction" relieves the THInC programmer of having to replace "2 *" with "2*" to optimise execution speed. Even if the "2" were hidden in a constant, THInC would optimise just the same.

If both parameters of the addition were literals, the sum would be computed at compile-time with no code generated at all. This "constant folding" relieves the THInC programmer of tricks like "... [constant1 6 * constant2 /] LITERAL ..." to prevent these calculations from being done at run-time.

These and other optimisation techniques reduce the code overhead to less than 20% over hand-assembled unstructured code and achieve a four- to six-fold speed increase over indirect threaded FORTH.

THInC is written in THInC so that although more computation takes place at compile time, the compilation speed is not degraded.

Other Features of THInC

In line with THInC's philosophy of giving the user as much programming assistance as possible, numerous error detection facilities are provided:

As in FORTH, mismatched or incomplete conditional structures are reported.

The pairing of ">R" and "R>" keywords is also checked to prevent program crashes due to imbalance of the return-stack.

The code of a definition is matched against its stack comment to detect any data-stack inconsistancies, a common programming error.

The user can control the system's reaction to detected compiling errors by setting the error-mode to either "abort", "warn" or "ignore".

At the same time a number of FORTH's restrictions were able to be lifted:

Conditional constructs can now be used outside definitions. Even if they are spread across several lines, execution is deferred until the construct is complete.

Also, conditional compilation is simply achieved by preceding the usual conditionals with a literal value, e.g.:

```
TRUE CONSTANT mode ...
mode IF ... ELSE ... THEN ...
```

The code within the "ELSE" clause is then not compiled.

To allow the programmer to access processor-specific resources, each THInC implementation has a number of nonstandard highlevel words, e.g.:

"int-on", "int-off" to control CPU interrupts;
"T-@", "T-!", etc. to access a microcontroller's timer.
"init:" marks the start of the program (similar to "main()" in C).

The objective is to enable high-level access to all processor resources. This provides the programmer with full control over the processor without having to revert to assembly language. THinC applications have been written without the use of a single line of assembly code. This even applied to the notoriously timecritical and difficult to debug interrupt routines.

Other features include an on-line screen-editor, a status line displaying the data-stack, vocabulary and other useful system information.

Application Experience

To date THInC has been completely implemented for the Z80 (THInC80) and 8048-family (THInC48) processors, the choice having been dictated by applications on hand rather than preference.

THInC80 is fully interactive system for the Z80 processor. A special effort went into efficiently simulating a second push/pull stack.

THInC48 is a cross-compiler as the 8048 processor family has separate code and data spaces [13] and therefore cannot be run interactively. It appears that THInC is the first high-level language to be available for this processor family. It allowed the software for a dual 8749-based medical cardiac monitoring product to be rewritten in record time while providing more user functions and exceptional ease of code maintenance.

The THInC (Threaded High-level Interactive nano-Compiler) system is based on the same successful interactive and extensible user interface as FORTH. This eliminates programmer retraining and allows well-behaved programs to be ported without modifications.

Future Developments

Versions for the 68000, 8051-family and 8086/PC are currently under development.

Further extensions of the THInC system may see data-type checking and the automatic conversion of data-types.

Conclusion

A code generating system has been presented that combines the acclaimed FORTH programming approach with the generation of highly optimised machine code. This combination could indeed prove a better "bridge" between the computing task and the processor hardware, thus resulting in higher programmer and system productivity.

It is thinkable that the efficient code generation and the availability of full processor control through processor-specific functions may render it superfluous and inefficient to revert to the use of assembly language when programming in THInC.

In contrast to traditional compilers THInC is user extensible in true FORTH style. THInC was not only developed to provide faster FORTH run-time execution, but also to allow the user to understand, maintain and modify a compiler.

Just as FORTH brought interactivity, modularity and extensibility to programming language design, THInC may bring these features to native-code compiler design.

References

[1] James J. S., "What is FORTH? A Tutorial Introduction", BYTE August 1980, p. 100-104.

[2] Dewar R. K., "Indirect Threaded Code", Comm. of the ACM, Vol. 18 (1975), p. 330-331

[3] Ritter T., Walker G., "Varieties of Threaded Code for Language Implementation", BYTE September 1980, p. 206-

[4] Sirag D. J., "DTC versus ITC for FORTH on the PDP-11", FORTH Dimensions Vol. 1 (1978), p. 25-29

[5] Delta Research, JFORTH User Manual, Palo Alto, CA 94306, USA

[6] Motalygo V. G., "PS - a FORTH-like Threaded Language", BYTE October 1981, p. 462-466; November 1981, p. 400-408

[7] Dowling T., "Automatic Code Generator for FORTH", FORML Proceedings 1981, p. 289-292.

[8] Langowski J., "Inline Code for MacForth", MacTutor August 1985, p. 59-63.

[9] Buvel R., "A Forth Native-Code Cross-Compiler for the MC68000", Dr. Dobb's Journal September 1984, p. 68-107.

[10] Greene R. L., "Faster FORTH", BYTE June 1984, p. 127-424.

[11] Novix Inc., Datasheets for the 4016, 5016 and 6016 FORTH Processors, Cupertino, CA 95015, USA

[12] Personal communication with J. Ogden during work on a FORTH machine-code compiler using the HP 64000 development system.

[13] Intel Corp., Microcontroller Handbook 1985, p. 12-1 - 15-42, Santa Clara, CA 95051, USA

About the Author

Klaus D. Veil has been utilising FORTH since 1981 in the hardware/ software systems he designs. Mayor projects have been a medical ambulatory monitoring system, a family of cardiac monitoring equipment and an industrial data acquisition and display system. Designing for equipment manufacturers has resulted in a focus on programming productivity and code efficiency.
He heads the Biomedical Research and Development department at the PACE Corporation in Sydney, Australia.

USING EXECUTION ARRAYS

&

TIME BASED OPERATIONS

IN FORTH

A Paper to the First Australian FORTH Symposium

D. C. Edwards - B. Eng. (Hons), M.I.E.Aust.

Director of Engineering

JARRAH Computers Pty. Ltd.

March 1988

Before I begin the paper, I'd like to give a brief background to my career as an electronic engineer. I worked in a research position at the University from which I graduated (U.W.A.) for six years and in 1982 I went freelance, finding work mostly in the music industry before attaining the position of chief engineer in a local electronic engineering company. It was in that position that I first used FORTH in an industrial control application.

Three years ago I formed Jarrah Computers as a consulting microprocessor engineering company and I have used FORTH as a key element in the planning and development of the company. Jarrah Computers has specialised in the development of custom designed industrial micro controllers - from the 68705 family of single chip controllers to education systems and large industrial control systems based on the Rockwell 65F11 - the 6502 implementation of a single chip FORTH.

FORTH has been used as the preferred development language for all of these applications and has provided Jarrah Computers with a consistent software environment for the wide range of development projects which it has undertaken.

The benefits to an engineering company of using FORTH in this way are numerous. We have found that software routines developed when writing code for one of the hardware systems can be used in many other places on different machines. This has meant that all contract work, custom developments and in-house developments have added to the Jarrah Computers pool of software utilities. If, during the development of a program, a way is seen to optimise or generalise the code of a Jarrah software module then it can be incorporated into the library of software utilities almost immediately.

In this paper I shall discuss some specific uses to which Jarrah Computers has put FORTH, namely the syntaxes surrounding the use of execution arrays and our development of a system for the coding of time based events.

As the themes of this Symposium are FORTH as a productivity tool and the impact of the NOVIX chips, I will address the productivity issue as an introduction to the body of my paper and will conclude the paper with a discussion of the impact and possible applications of the NOVIX.

FORTH AS A PRODUCTIVITY TOOL

The business of writing software can be summed up as:

Writing the text which can be sent to a compiler which uses that text to produce the desired program.

The choice of a language is largely the choice of which "high-level" text phrases you prefer to write. The "power" of a high level language is measured in terms of the ratio of the amount of code compiled to the number of language words invoked to produce that code.

For example, languages such as BASIC, C, and Pascal provide compiling facilities which allow the programmer to pass multiple arguments within complex expressions, define field types, define sets (with automatic checking included) and so on. These facilities **are designed to produce greater programmer productivity** and FORTH is often criticised in this respect - that functions normally expected of the compilers of other languages are not provided in FORTH.

However, FORTH has a facility that allows it to overcome the apparent limitations of the simplicity of its own compiler - the Compiling and Defining words. These words allow the programmer to create entirely new compiling facilities and new high level syntaxes.

The compiling facilities of FORTH can be even more powerful than those of high level languages because they can be tailored to the task at hand. In FORTH, one uses the idea that a few compiling words are as much a part of the application as the actual compiled code. The irony of the promise of high level languages is that one ends up fitting the desired compiling facilities into those provided in the language - the "power" features can often be a hindrance and are sometimes irrelevant to the task.

When examining the question of software productivity, it is important to remember that the business of creating software literally means **writing** software. Software is a form of literature and much of the work is similar to the work of writers of literature - sitting at a desk (or word processing terminal), thinking and writing. And the productivity issue gets down to the faster the software can be written, the more productive the system. This means that not only must the program design and the management be efficient, but equally important the physical act
of writing it down must not be too cumbersome.

Through the Compiling and further through the Defining words, the writer is provided with an unparalleled opportunity to control what actually needs to be written. If you write a compiling utility which allows you, for the rest of the application, to describe things in a few words rather than a few paragraphs, there is an enormous increase in productivity. Not only are the editing sessions easier, all aspects of the writing process are simpler - the document is smaller, printouts are easier to scan etc.

The trade-off, in the productivity issue, between FORTH and other languages is that in FORTH, all utilities must be developed, whereas with other languages many utilities are provided as part of the purchased package. To decide whether FORTH will provide productivity benefits in a particular application is a difficult assessment, but in general FORTH is most productive in a development environment where the tools required may not even exist yet, let alone be available in some off-the-shelf package.

For instance if one is developing a data base application, a pre-written package will allow you to start writing the application immediately. With FORTH, a set of primitives which perform the file interfacing, menu generation, query handling etc. would have to be developed. An experienced FORTH programmer, who had developed these tools, would be in a similar, if not **better**, position than the package user, because they not only have the functions needed, but those functions can be easily tailored to the specific job. FORTH is like a financial investment - it may require a large contribution of time and effort at the beginning, but the dividends will continue for some time.

My approach to writing software is what I refer to as "syntax driven" programming, where the desired syntax (the "what-you-want-to-write-later") is used as the basis of the design and implementation of the software, and I have found it to be effective. Syntax driven programming has manifold benefits both in technical execution and in project management.

Technically, it forces the words to have a consistent use, long before they are written. Various different applications of sets of the words can be tried to see if the planned syntax is correct and that words work in different contexts. Through refinement, the syntax becomes the data flow diagram, the module specification and the implementation schedule all in one. From a project management point of view, each member of the development team is aware of the overall system (as there are always only a few words at the heart of a system) and benefits flow from their interchanges.

The approach also leads to many good programming habits, principally good naming and information hiding. An example of good naming syntax is with my experience of finding a name for a word which changes the current file of a resident DOS.

My first response to the name was "SET-CURRENT-FILE-TO", which I then shortened to "SET-CURRENT-FILE" and then to "SET-FILE". Only when I discovered the preferred name, which is **USING**, did I realise why it was better - it's better in the overall syntax. In fact, it CREATES a syntax. To demonstrate the advantages of USING, the problem with SET-FILE is that it is ambiguous as it could work in two different contexts, either

```
(handle-parameters-on-stack)  SET-FILE
```

or

```
SET-FILE  MYFILE
```

whereas

```
USING  MYFILE
```

is never ambiguous because one tends to think the **phrase**

"USING MYFILE".

Information hiding almost naturally develops out of syntax driven programming because names come first and the specification of the named words comes later. I have taken an example of the benefit of this information hiding from a program which had to decide whether to move a vent. The phrase I conceived was something like:

RANGE OUTSIDE-OF IF MOVE.VENT THEN

read as "If you're OUTSIDE-OF the RANGE of temperatures which is allowed at the moment, then MOVE-VENT. The argument plan was:

RANGE would push values,

OUTSIDE-OF would test these against the current temperature and set a flag for the IF clause if movement was required.

IF MOVE-VENT THEN conditionally performs the action.

After implementation it turned out that all RANGE did was C@ twice - to fetch the upper and lower limits. I almost dropped it as a definition at this point.

Later, it transpired that the algorithm needed further sophistication and RANGE turned out to be the word which could provide the necessary extension. RANGE was then developed as a word to fetch two numbers from a complex number array, using a very complex calculation to even work out the index into the array. During the development of this word, no integration difficulties were experienced because during and after the changes RANGE always returned two numbers on the stack. All of the words which expected RANGEs output worked after the change as before.

This late (and very complex) change would not have been so easy if I had dropped the original definition of RANGE - if I had not "hidden" the information from the system behind the definition.

The software I will outline in this paper has all been developed using this syntax driven approach. From the beginning of my experience of writing menu interfaces for industrial controllers, I knew I wanted to write something like

THIS.MENU DOES (unrestricted FORTH)

(and its) KEYS (unrestricted FORTH)

to describe the operation of each menu. And I didn't want to have to remember numbers associated with each menu, or set up execution arrays. That syntax was ALL I wanted to write! Nine FORTH screens later, I had the syntax at my disposal. Similarly in the timing system, the impetus was to declare time periods as

and have words like AFTER and FOR which followed them. Again, I didn't want to have to worry about where the current time (for this or that process) was being stored, how time information was passed around and so on. And again, another ten screens later it was implemented. This is how FORTH can truly increase productivity.

EXECUTION ARRAYS

Execution arrays use vectored execution, which refers to the technique of storing the address of a routine in a VARIABLE. To run the routine, the content of the VARIABLE is fetched and executed. In FORTH this requires the phrase:

```
VARIABLE-NAME @ EXECUTE
```

Execution arrays are a set of vectored execution points, collected together in an array. The ARRAY-NAME points to the base of the array, so to execute the nTH element of an array we use the phrase:

```
2* ARRAY-NAME + @ EXECUTE    ( n --- )
```

I have found the word @EXEC useful in executing vectors in such arrays. It expects the index into the array and the array base address on the stack and executes that element of the named array.

```
: @EXEC                                    ( n A ---- )
        SWAP 2 * + @ EXECUTE ;
```

The address stored in the VARIABLE (or array) points to the beginning of the CODE (the run-time colon code) of a FORTH word. To create and use execution arrays requires the programmer to explicitly find out the addresses of the words which are to be inserted into the array and store them into the correct locations in the array.

This can be done with the phrase

```
' NAME-OF-ROUTINE   Index-of-array 2 * ARRAY-NAME + !
```

This phrase must be repeated for each entry in the array.

For the Jarrah systems (logging and menu generation) I have developed syntaxes which automatically perform the work involved in stuffing the addresses of the executable Colon definitions into the arrays. The syntax involves two steps:

1. Naming the loggers/menus.

2. Defining the operation of each logger/menu.

Both the menu system and the logging system have two functions each - the menus have a display action and a keyboard response action, and the logging

system has the log action and the display action. Accordingly two arrays are set up for each.

Logging arrays: 'LOGS and 'SHOW
Menus arrays: 'DOES and 'KEYS

These names are always analogous to the keywords used to define the functions and use a naming convention proposed by Leo Brodie in "Thinking FORTH" - the "'" is short for "address-of".

The logging system syntaxes developed as follows. Firstly, for the logging actions, the aim was to have a syntax of the form

TYPE-OF-LOGGER LOG

(for example AVETEMP LOG or SENSORS LOG etc.)

and secondly, for displaying data out of the log, the desired syntax was

(ADDRESS WITHIN LOG) SHOW

I decided to trial the system presuming that the TYPE-OF-LOGGER word would simply push a number. This means that the word LOG must be the point of vectored execution (using the 'LOGS array) - it uses the number pushed onto the stack by the name and does the specific logging action for that name. As the log type will itself be logged (so SHOW knows how to display the logged data), LOG must be something like:

```
: LOG
     DUP LOGC,
                              ( common logging )
     'LOGS @EXEC              ( specific part of logging )
                              ( common work after logging )
;
```

Common logging actions are, for example, log the time, date etc. and the common work after logging is adjusting pointers, testing for wrap-around etc.

(This is another benefit of the execution array - that any generic code needed in any of these words (setting up a menu or performing the key handler or the logging or showing part of a logger) can be included in the definitions of the vectored execution words.)

Moving on to how SHOW must work, it has an address on the stack when it runs, so it must fetch the type byte out of the log and use it to execute the display action for that particular logger. So it again must be the point of vectored execution (using the 'SHOW array), and must be something like:

```
: SHOW
          LOGC@
                              ( common showing )
          'SHOW @EXEC         ( specific part of showing )
```

Having decided that this was a workable design for the names of each log (i.e. to simply push a type number onto the stack), the design of the naming process is almost trivial. A temporary VARIABLE is used to hold the next logger (or menu) name number. Each time a new functional is defined, a CONSTANT of the value of the contents of the VARIABLE is created and the VARIABLE is incremented. For the logging system, the word LOGGERS: puts a zero into the variable and LOG: is the name creating word. This is how the naming process looks in its final form.

```
LOGGERS:

LOG: AVETEMP   LOG: CONTROLSTATUS  LOG: SENSORS

;LOGGERS
```

The next question was how are the logging and showing actions for each logger to be specified, and their addresses inserted into the correct entries in the relevant arrays. Again the desired syntax was something like

```
NAME     LOGS:    (standard FORTH)
         SHOW:    (standard FORTH)  ;
```

Having proceeded this far with the syntax, I could deduce what LOGS: and SHOW: as words must do. LOGS: must

1. Fetch HERE and store it into the current index in the 'LOGS array.
2. Compile the run-time colon code.
3. Set the FORTH machine into the compile state.

and SHOW: must

1. Compile the run-time semi-colon code (to end the LOGS: definition)
2. Fetch HERE and store it into the current index in the 'SHOW array.
3. Compile the run-time colon code.

These words are now well enough specified to be written. Note that their specification has been arrived at by following the course of what they MUST do to make the SYNTAX work. Note also that using such a system, no headers except those used for the names of the menus are generated.

The following example shows the system in its completed form.

```
CONTROLSTATUS

LOGS:       CONTROLSTATE   C@ LOGC,
            ERRORTEMP      C@ LOGC,

     SHOW:  LOGC@ CASE
       0 OF ." Control Monitor"              ENDOF
       4 OF ." Heat Monitor"                 ENDOF
       5 OF ." Heater ON"                    ENDOF
       6 OF ." Heater OFF"                   ENDOF
       8 OF ."VENT ." ing" "STARTED          ENDOF
      10 OF ."VENT ." ilation " "CYCLE       ENDOF
            CASEND
            LOGC@ .ERRORTEMP ;)
```

And finally, to perform the logging, all that is needed is

```
CONTROLSTATUS LOG     AVETEMP LOG     etc.
```

The menu system followed a similar development, although it was much more complicated as number handling was included as a part of the automatic processing, and here is an example of it in its final form.

```
MENUS:
MENU: MAIN          MENU: PROGRAM       MENU: CONTROL
MENU: MANUAL        MENU: EDIT.CYCLE    MENU: SET.TIME
;MENUS

PROGRAM MENU
        DOES:  CLS    LN1 .DATE .TIME
                      LN2 ." 1> Time 2> Cycle 3> Control"
        KEYS:  CASE       "1" OF SET.TIME          ENDOF
                          "2" OF EDIT.CYCLE        ENDOF
                          "3" OF CONTROL           ENDOF
                          "P" OF MAIN              ENDOF
               CASEND >MENU ;
```

This concludes the explanation of the syntaxes which Jarrah Computers has developed to facilitate the use of execution arrays. I think this compiling utility is a good example of the FORTH-as-investment idea which I mentioned before. It took considerable work to develop the system, so the "productivity" was low during this phase.

Now that it has been developed, however, Jarrah Computers can write a menu structure within hours of the design being completed - in fact the system has been evolved so that programmers can sit down with the menu designs in front of them and key the menu code in straight from the specification. The boring (and therefore error prone) editing sessions and index tables needed to

code the menus without the aid of the system can be discarded and programmer productivity is now higher than was ever possible before.

Its harder to assess the increase in productivity which flowed from the development of the syntax for logging, because I never coded logs in any other way - this was my first solution. Its simple to use, easy to modify and occupies less than 800 bytes of code space.

TIME BASED OPERATIONS IN FORTH

The Timing system starts with what have been dubbed the TIME VARIABLEs (TVs) - named SECS MINS HRS and DAYS. These words push indexes onto the stack - they are numeric names for each of the time-period entities. The implementation of the Time Variables depend on what is being used as the basis for the timing (usually either a Real-Time-Clock or an interrupt), so their implementation will vary, but **they always push a number onto the stack**. The TVs may range from being the names of standard FORTH VARIABLEs to being offsets into an array (similar to FORTHs USER VARIABLEs).

Subsequent to the definition of the Time Variables, a word called @TIME is then defined which expects a Time Variable on the stack and returns its current value. This word has obvious application and it also aids the transportability of the system as it provides a level of information hiding. It is usually a small stub - for example, in the two applications I have implemented, @TIME was defined as C@ or @RTC respectively.

I have found it useful to define the words:

```
: @SECS         SECS @TIME ;
: @MINS         MINS @TIME ;
: @HRS          HRS  @TIME ;
```

as they are used so often it saves space and also they are more readable.

The next word in the lexicon is NEW, which expects a TV, along with a previous value of the same TV on the stack. It returns both the new value of the TV and a flag on top of it indicating if the value was different (i.e. if there is a NEW value). NEW is therefore the first word which detects time based events - it compares two different values of a single Time Variable and produces a flag which represents information about time.

```
NEW                         ( n TV -- n' Flag    )
                            ( Flag True if n' not = n )
```

Note that NEW can be run continuously and will only return a true flag the first time it detects a new value of the TV. As NEW returns a flag, it is designed to be followed with a conditional branch in syntaxes such as

```
                ( ... @SECS ... )
and then        SECS  NEW IF .... ELSE ..... THEN
```

and @HRS
BEGIN PAUSE ..... HRS NEW UNTIL (DROP)
etc.

The PAUSE after BEGIN is the multitasking link word which ensures that other tasks will run while the above process is waiting to terminate. The multitasking system used is a FORTH convention (if not standard), and the only word which is overtly used in the Timing lexicon is PAUSE, which stops the current task, proceeds through the round-robin of other tasks and finally returns to the task in which the PAUSE was located. Some understanding of how multitasking works is necessary to understanding the application (if not the coding) of the Timing system.

The Time Variables, @TIME and NEW are enough to implement time based regimes of operation. For instance, in the outer loop of a task where it is desired to perform an action every second (called EACH-SEC), the invocation is

```
: TASKSETUP        TASK-NAME ACTIVATE
                   @SECS
      BEGIN SECS NEW IF EACH-SEC THEN PAUSE AGAIN ;
```

Note that the current (or last) value of seconds is **always** on the stack of this particular task. This is a deliberate feature of the design, so that temporary VARIABLEs and period VARIABLESs are not needed.

Similarly, tasks can be set up to execute EACH-MIN and EACH-HR routines, or one task can handle them all:

```
: TIMINGSETUP
      TIMING ACTIVATE
                              @HRS @MINS @SECS
                  BEGIN SECS  NEW IF EACH-SEC  THEN PAUSE
                  SWAP MINS   NEW IF EACH-MIN  THEN PAUSE
                  ROT HRS     NEW IF EACH-HR   THEN PAUSE
               SWAP ROT AGAIN;
```

The arguments provided to NEW are enough to specify a time event. To specify a time period, we simply need to add a count to the events. This is achieved in the syntax

n TV

e.g. 30 SECS 4 MINS 2 HRS etc.

The next word in the lexicon is ELAPSED, which is similar to NEW in that it returns a flag if the specified period of time has elapsed. Its detailed processing is complex so I will just say that it uses @TIME to calculate the net elapsed time (since its last invocation) and this is subtracted from the count to produce the flag.

ELAPSED	(c TV n -- c' TV n' Flag) (Flag True if c' < 0)

To move from this blow by blow description of the words in the lexicon to a more general description of the aims of the system, the types of functions which industrial control programmers need, and the programming of which I was attempting to simplify, are things like:

1. Wait for specified period.
2. Wait for specified period OR until condition occurs.
3. Wait for specified period and TIME how long it takes for condition to occur.
4. Perform function until Time-of-Day = Specified time.
5. Perform function periodically, for X times.
6. Perform function periodically, terminated by Time-of-Day = Specified time.
   etc.

After considerable analysis it transpired that all of these functions could be expressed in three main syntaxes:

(PERIOD) FOR CONDITION MONITOR

(PERIOD) FOR CONDITION DETECTED
IF ... ELSE ... THEN

(PERIOD) FOR CONDITION TIMED
IF ... ELSE ... THEN

All of these constructs use the same initial syntax (a period is specified and then the word FOR is stated), but they use different terminating words - MONITOR, DETECTED or TIMED. Their actions are similar in that they will all wait in a loop and terminate when either the specified period has timed out or the condition clause has left a true flag. The difference between them is what is left on the stack at the end of the process:

MONITOR	leaves nothing on the stack.
DETECTED	leaves a flag which is true if the condition occurs and false if the loop timed out.
TIMED	leaves the same flag as DETECTED, and the time it took for the condition to occur will be left under a true flag.

FOR, MONITOR, DETECT and TIMED are compiling words and they all compile various words around and inside a BEGIN ... UNTIL loop. FOR compiles BEGIN, PAUSE and ELAPSED and the terminating words compile OR and UNTIL and various stack manipulators to provide the final stack values.

As an example of the uses of these syntaxes, the word AFTER can now be easily defined as:

```
: AFTER        FOR 0 MONITOR ;
```

This is another way of writing MONITOR nothing, and it forces a full wait for the period specified before AFTER, as the condition clause (0) is always false.

The four words represent different levels of timing information:

AFTER simply consumes the specified period of time.

MONITOR will use up the time or break if a condition occurs.

DETECTED lets you know if the condition occurred or not.

TIMED lets you know if the condition occurred, and also provides the amount of time that elapsed before it occurred.

I have found when developing time based algorithms, that the control words can be developed from a simple "get-it-running" solution into a more "intelligent" or "adaptive" program by utilising these different control structures at different points in the development. For example, this heating algorithm

```
: HEAT
                    CLOSE-HOUSING
                    PHASE# 2 >
          IF 180 SECS FOR HEATED COLDER OR MONITOR THEN
                    HEATED NOT
          IF HEATER ON (Do heating process .....)
     THEN ;
```

performs a wait before evaluating whether to turn the heater on.

In my first version of the code, I simply used

```
: HEAT
                    CLOSE-HOUSING
                    PHASE# 2 >
          IF 3 MINS AFTER
     THEN .....
```

Then I realised that it could be made smarter to break if either the environment had heated up (in which case no active heating would be required) or it had become even colder (in which case, it should stop waiting and start the heater immediately). The FOR .... MONITOR construct followed by the HEATED NOT test provides all of these functions in the most efficient manner - FOR ... DETECTED could be used, but more (redundant) flag checking would have to be done.

The next example, using the DETECTED syntax, provides a demonstration of using the Timing words inside a standard FORTH DO ... LOOP construct.

```
: DO-WATERING
      #VALVES 0 DO   NEXT-VALVE ON
                     LAST-VALVE OFF
      @VALVE-TIME    FOR OVERPRESSURE  DETECTED
                     IF OVERPRESSURE! LOG THEN
                LOOP LAST-VALVE OFF ;
```

This algorithm is a functional description of the control algorithm of the first major program I ever wrote. That version used VARIABLEs in which to hold step counters, the amount of time-left for the step, pointers to first and last valves and so on. The above version uses NO variables, uses one phrase of the Timing syntax and a single FORTH DO ... LOOP. And it does the control process better!

Just as any AFTER period can be improved by replacing it with a FOR ... DETECTED loop, so a FOR .... DETECTED loop can often be improved by replacing it with a FOR ..... TIMED loop. For example, in an a control application where a motor is moving a physical device, the job often reduces to

ActivateMotor

Wait-for-Sensing-Switch-Closure (or Opening).

De-ActivateMotor

This is a classic application for FOR .... DETECTED:

```
                                MOTOR ON
10 SECS FOR SWITCH.CLOSED DETECTED NOT
                                MOTOR OFF
                    IF MOTOR.ERROR LOG
          THEN ....
```

Note that NOT can be used freely in the syntax - one can write SWITCH.CLOSED NOT DETECTED and so on. Also note that any actions are allowed after DETECTED and before the IF ... ELSE .... THEN clause - in the above example it is important to turn the motor off immediately that the loop terminates.

It then occurred to me that the controller could measure the amount of time conditions took to occur after actions, and then build up averages of these numbers. Through this method of data collection, the controller can develop an idea of the typical performance of the physical system to which it is connected. An example

```
                    MOTOR ON
10 SECS FOR SWITCH.CLOSED TIMED
                    MOTOR OFF
       IF Use Number to update AVERAGE
ELSE MOTOR.ERROR LOG
THEN
```

The final aspect of Timing which I will address is a few words to allow the above constructs to incorporate statements about the Time-of-Day. This lexicon uses a key concept used in the first FORTH timing system I ever encountered (Bill Ragsdale's article in FORTH Dimensions Volume 5 Number 5), the idea of storing real time as a single precision number, representing the current Minute-of-Week, which he called MOW. Whilst this idea can be extended as far as a Minute-of-Month, I use a cut-down version of his system, the Minute-of-Day (which I call MoD) as this is the most appropriate for a system based on a real-time 24-hour clock.

The advantage of converting the Hour Number and the Minute Number into a single precision number on the stack is that comparing times and calculating differences between times etc. can all be easily performed using FORTHs standard arithmetic operators. (-, +, <, & >.)

The words I use are:

```
: >MoD                          ( #HRS #MINS  -- MoD )
        SWAP  60 * + ;

: @MoD                          (    -- Current MoD )
        @HRS @MINS >MoD ;
```

PAST and BEFORE both expect a Time-of-Day as two numbers on the stack (Mins on top) and compare this time with the current Time-of-Day. PAST returns a true flag if the current Time-of-Day is greater than or equal to the stated time, and BEFORE returns a true flag if the current Time-of-Day is less than or equal to the stated time .

```
: PAST                          ( #HRS #MINS -- flag  )
        >MoD @MoD > NOT ;

: BEFORE                        ( #HRS #MINS -- flag  )
        >MoD @MoD < NOT ;
```

BETWEEN expects two Time-of-Day s on the stack and returns a true flag if the current Time-of-Day is between the two input times.

```
: BETWEEN
            >MoD >R >MoD R> 2DUP > @MoD SWAP
        IF SWAP OVER < NOT SWAP > NOT OR
    ELSE WITHIN
    THEN ;
```

These words allow statements like

```
BEGIN PAUSE ..... 18 30 PAST UNTIL
```

which will remain in the loop until 6:30 P.M., and

```
3 MINS FOR VERY.HOT
          LATE.CYCLE AND
6 00 15 00  BETWEEN   AND DETECTED
      IF .... ELSE ... THEN
```

a condition clause which will be detected only if the conditions VERY.HOT & LATE.CYCLE are met and its between 6 A.M. and 3 P.M.

The logging syntaxes described earlier and the Timing syntaxes work very well together. Logging is often based on Time, with the logging requirements often generically stated as

```
log THIS, every X period for N readings
log THAT, every Y period for M readings
```

and so on. This can be easily accomplished by using AFTER and LOG in a DO ... LOOP construct. For example, to log the tire pressure every 30 minutes for 50 readings,

```
50 0 DO TIREPRESSURE LOG 30 MINS AFTER LOOP      etc.
```

To log something every so often until a specified time-of-day rather than for a specified number of readings, we use AFTER and LOG inside a BEGIN .... UNTIL loop, for example to log the average temperature every 2 minutes until 6.00 P.M.

```
BEGIN  2 MINS AFTER  AVETEMP LOG  18 00 PAST UNTIL
```

These timing words are not restricted to the above syntaxes and can be used in free-standing FORTH expressions. For example, the time-variables specify the time scale which a process uses. If for instance 1 MINS is specified, then on the next change of minute, a process will terminate. If it starts at the 58th second of a minute, the total period waited is 2 seconds. To be able to specify a minute as 60 SECS, or an hour as 60 MINS, I developed the words HOURS and MINUTES, defined as

```
: HOURS          60 * MINS ;

: MINUTES        60 * SECS ;

used as      3 HOURS   FOR .....

and          15 MINUTES   FOR .....      etc.
```

(The naming idea here was that writing the name in a higher resolution (HOURS instead of HRS & MINUTES instead of MINS), generates a timing loop of consequently higher resolution.)

As another example, I have found it useful in some cases to add a guard clause to the beginning of a FOR .. DETECTED loop, to prevent it breaking out of the loop for a certain minimum time. The word GUARDED suggested itself, used as:

30 SECS 10 GUARDED FOR ....

which would wait for a total of 30 seconds, but would "guard" the first ten seconds against the break. GUARDED simply compiles an AFTER of the period specified and subtracts this period from the total period, ready for FOR. In the above example, it would compile code equivalent to:

10 SECS AFTER (30 - 10 =) 20 SECS FOR .....

and can be defined as

```
: GUARDED
        ROT OVER - ROT ROT OVER AFTER ;
```

As a final example, the word TILL can be easily defined, to allow Time-of-Day specification of the end of a FOR loop, used as

15 00 TILL ..... MONITOR etc.

and defined as

```
: TILL
        >MoD @MoD - DUP 0< IF 1440 + THEN MINS
        [COMPILE] FOR ;
```

To conclude this section, the Timing words outlined in this paper:

SECS	MINS	HRS	DAYS
@TIME	NEW	ELAPSED	
FOR	MONITOR	DETECTED	TIMED
PAST	BEFORE	BETWEEN	

allow description of all of the "desired" process control functions listed above, and they cover the many different ways of talking about time - in terms of elapsed periods, in terms of Time-of-Day, in terms of "when condition occurs" etc. They form a consistent, and re-usable lexicon.

The timing system is less of an example of using FORTH for productivity gains, than an example of how the total lexical freedom possible in FORTH allows the programmer to describe the problem exactly as desired, unfettered by any a priori high-level "rules" or "power" utilities. This is a **language** of time within FORTH, which can express time logic, manipulate time periods and execute time-based actions.

I will conclude this paper with a discussion of the implication and applications of the NOVIX chip as I see them.

THE IMPACT OF THE NOVIX
&
SOME AREAS OF APPLICATION

IMPACT

A processor of this net execution speed is obviously going to find and even attract many applications. The machine is already providing not only a very fast working environment, but also a general focus for those interested in FORTH. It is galvanising a lot of effort behind the language and will continue to do so.

I think the NOVIX will attract all people interested in fast processors, in particular, the writers of assemblers and compilers. This means that a lot of people outside of the FORTH community will get to know the language. Furthermore, I believe that there is a good chance some of these people will be converted to FORTH by the NOVIX, and so the general FORTH community will grow.

As well as the sheer execution speed of the processor, the NOVIX has other attractive features - the square root function, very fast multitasking, and the optimising compiler to name but a few.

The optimising compiler means that FORTH text is compiled into very fast, compressed code and, according to all accounts, it generates truly optimal code. This is why, when Daniel L. Miller came to write an implementation of the C language for the NOVIX (see BYTE April 1987 p 177), he did not write a C compiler which directly generated native NOVIX opcodes, but chose to write a program which converted C **text** into FORTH **text**. The resulting FORTH text file is then fed into the optimising NOVIX compiler. This turned out to be the most efficient way of generating the target code.

As he points out, not only does this code run very quickly on the NOVIX, but a new level of transportability has been added to the C code - at the FORTH text level the code can be fed into ANY machine that interprets and runs FORTH. This feature will again attract many people into the FORTH environment that may not have been previously interested, and the transportability of the interstitial text files is a benefit in itself.

APPLICATIONS

1. Hardware-in-Software

The potential areas of application of the NOVIX range of chips are numerous. The first area of exploration must be the concept of executing hardware systems in software. The application of the NOVIX may be similar to the Transputer in this respect - it has found much work in the software-doing-the-job-of-hardware driver area. (Graphics engines and Disk controllers).

The floppy disk controller provides a good example of this. In this interface, the use of Timer chips for various wait times and hardware to wait-for-condition-and-set-bit is standard. The promise of the NOVIX in this area is that ALL of this work could be done in software - the timers, the detectors and so on. The NOVIX promises even more power, through its ability to swap between controlling different types of drives by simply changing ROMs (or even just VOCABULARYs). Through the use of multitasking, it should be possible to control different drives simultaneously.

Other areas of hardware research in which the NOVIX could play a part are:

1. **Digital Signal Processing** -This was one of my first interests in digital computing. I had been involved in analog processing for some time, and only with the release of the Texas Instruments 32010 did I decide to "convert" to digital.

   However, on detailed inspection, although the 320 could do an accumulating multiply in 200 nS, it had drawbacks which rendered it almost unusable for the applications I was envisaging. These drawbacks included such things as the Harvard architecture, which makes immediate loads impossible.

   When I looked at trying to design a full system on a 320, the tasks of handling (even minimal) keyboard inputs, scanning (even minimal) timers or interrupts and so on would take so much time that the processing power of the device was lost. The 320 is not that good at anything BUT accumulating multiplies in 200 nS.

   The NOVIX however is the reverse case. It takes about 2 MICRO seconds to do an accumulating multiply, but everything else in the processing can be handled quicker. The speed of the machine promises the ability to write keyboard scanners, hardware polling routines and so on, all without disturbing the timing of the other ("main") tasks.

   The potential of having a MultiTasking system running fast enough to control hardware is tremendously exciting. In a disk drive interface, the 555 timer hardware could be implemented as a TASK, with another major task setting the functional tasks in motion, and waiting for their termination and so on.

2. **Network controllers and protocol converters** are another area for application of the same basic idea. Again the ability of the NOVIX to set up an analog of a hardware system (in digital!), and the ability to switch between different flavours of hardware simulation with the speed of setting a new execution vector, or a different vocabulary, is the key feature of the NOVIXs in this application. A device which can interface and convert between different machine protocols and which can be changed with the ease of a re-edit (rather than a re-build) must be attractive commercial property.

2. Processor Simulations

Ever since I first wrote a time critical piece of assembler code and had to count the cycle times of every instruction many times, I have had a peculiar sensitivity to how many machine clock cycles it takes a processor to execute its opcode. This is why I was attracted to (and still use) the 6502 - most instructions take 3 or 4 cycles, some take 2 and some take 5 to 7. When I was asked to convert my code onto another processor, I couldn't find one at that time which would end up running it anywhere near as fast (including the Z80, 6809 and 8086). When I read the clock cycle times for the 80186 and the 68000, I laughed - microcode was getting out of hand. One instruction takes 135 cycles! This is obviously running a small program on-board. The point of the 6502 is that a sub-routine running a couple of dozen opcodes will run faster than a lot of **instructions** on other processors.

The NOVIX is, quite frankly, the **answer** to this problem. The first few layers of definitions in a NOVIX application (for example : OVER >R DUP R> SWAP ;) are effectively new opcodes. These definitions takes the place of the standard MicroCode. These words take only a few clock cycles to run (5 in the above example) and so hand tailored "opcodes" can be written.

As an example of this idea of switching between different types of operation with the flick of a vector, I have considered the idea of writing NOVIX programs so that the machine would execute the native opcodes of various processors. We could have VOCABULARYs named 8088, 68000, 6502 etc. Given the clock cycle consumption of these chips mentioned above, this idea should be entirely possible and it would be interesting to see how fast the simulations run. This general idea suggest applications of the NOVIX in processor research, development and simulation.

3. Industrial Controllers.

Modern industrial control has two levels of processors - so called Field processors and Control processors. The field devices handle sub-sections of the plant and/or perform specialised analysis of a specific process, and the Control processors handle operator interfaces, alarm processing, trending and complex mathematical control process modelling to determine optimal loop tuning.

The trend in industrial control at the moment is towards more processing power at lower levels in the system (as in most areas of this industry), and more "expert" judgments are being expected of not only the main Control processor, but also of the Field controllers.

The NOVIX presents itself as a candidate in **both** of these areas - its speed of multitasking makes it an ideal field instrument, and its sheer computing power makes it at least considerable as a solution to the Control computer job. Future application software developments will determine how far the chip makes it into this area.

4. Multiprocessing.

I think that the most important aspect of multiprocessing is architecture, because it underlies the whole machine and it often forces the software into certain a priori structures. A highly desirable feature of the processors in such an environment is minimum chip count, because as the number of processors increase, the hardware and the support circuitry (power supplies, connectors etc.) can expand beyond feasibility. This thinking underlays the design of the Transputer - only memory chips need to be added to the transputer chip to get a functioning unit complete with four bi-directional serial lines and so on.

The NOVIX, in contrast, is a processing engine - it is a bare CPU in terms of hardware functions and support. This is the basis of its processing power but it makes it more difficult to implement as a sub-unit. It has no interrupt priority handlers, timers, communication channels etc. as part of its make-up. This means that boards incorporating a NOVIX must have quite a large amount of support circuitry.

It is interesting to compare the NOVIX to the 65F11 in this respect. The F11 is based on the 6502, an old processor, but what made it attractive from a hardware development point of view is its on-chip features (FORTH kernel, serial channel, 8 level interrupt handler etc.) and a minimum hookup of five external chips. I have considered quite a few different designs for multiprocessing using the F11 and they are feasible due to these minimal support hardware requirements.

An idea which keeps going through my mind in this respect is to get the stacks on-chip. This would be a major leap forward in terms of using the NOVIX in a multiprocessing environment.

That much said, multiprocessing is certainly possible on the NOVIX. The ports could be used to great advantage, especially to implement banks of memory. These could be a useful expansion for the NOVIX in any case and they would be very useful in applying the NOVIX to multiprocessing. Multiple machines, each running multiple memory banks, with some common banks for communication and data passing, is, I feel, the most viable approach to a hardware architecture for multiprocessing with the NOVIX.

From a software point of view, the NOVIX is a very interesting candidate on which to develop a multiprocessing operating system. Each processor could have a common kernel (say the 8 K system now) and so could make calls to each others kernels very easily (via a two-byte address packet!).

Again low-level high-speed multitasking is critical to this implementation - without it, it would be very cumbersome to even build an architecture. Each multitasking processor could have one task dedicated to communicating to the other processors and through this connection the various processors can co-ordinate to become a single bigger machine. In this respect I think the NOVIX is in a better position than the Transputer which is limited to a certain extent by using hardware to perform the communications.

To conclude my paper, I would like to thank all of the people involved with the organisation of this first ever Australian FORTH symposium, and all the delegates who have attended. I hope we can use this opportunity to get to know one another and develop the basis for a strong, interactive (and hopefully highly productive) Australian FORTH community.

Finally, I would sincerely like to thank Charles Moore, firstly for attending our symposium and providing us with the benefit of his experience and unique insights, and also for inventing FORTH.

Forth Engines vs Assembly

(Forth Applications in industrial control)

by Ray.Gardiner
Ardmona Fruit Products Co-op Co Ltd.
PO Box 196, Mooroopna 3629
Victoria, Australia.

ABSTRACT

Programming real time control systems in Forth allows the interactive development and interactive debugging of hardware interfaces. The paper explores the issues of development techniques, software maintenance, execution time speed and complexity.

Introduction.

The target audience for this paper is the software/hardware applications programmer who is interested in using Forth for real time process control or machine control applications.

There is now available a full range of development environments for generating Forth based applications these include many multi-tasking and multi-user systems. As well as simple low cost single board systems. Most applications can be prototyped and tested interactively, allowing the programmer to concentrate on solving the problem.

The traditional trade-offs involved when comparing assembler with high-level languages can be discounted completely with the availability of 16 bit and soon 32 bit Forth engines.

Designing and writing an industrial control system with Forth is fast and easy when compared with other available techniques.

The factors to consider when choosing a language for real-time control systems.

1..INTERACTIVE DEVELOPMENT AND DEBUGGING.
(the development cycle)

2..MAINTAINABLITY

3..DOES THE LANGUAGE HELP TO DEFINE THE SOLUTION?.

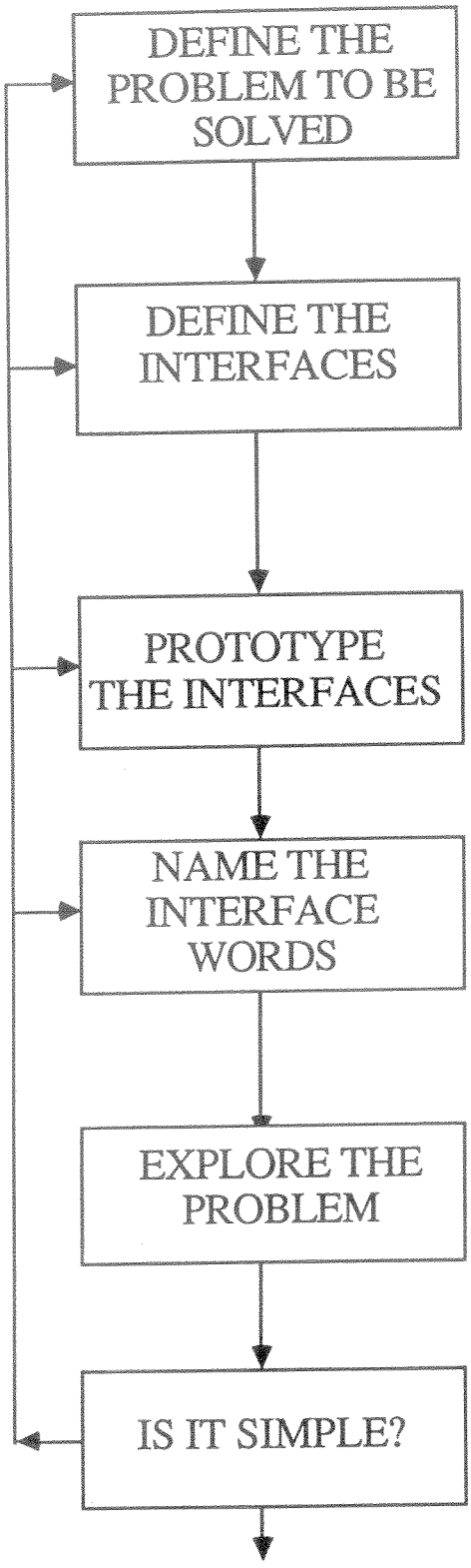
No, I haven't forgotten execution time speed. Forth engines have rendered this subject obsolete!

For the sake of completeness, there is a special section of the paper devoted to discussion of execution time speed.

The ability to mix Forth and assembler to achieve desired performance levels is essential where conventional processors are used.

Application development cycle.

The steps involved in writing the application are usually as follows.



It is quite normal here to be given the solution rather than to be told what the problem really is.

Are you fixing the results of another problem!.

What does your program have to talk to?
Is there an operator?. What is the interface to the operator?.
What are the I/O devices. Coils/Sensors.?
What are the timing contraints?
(how fast? and how often?)

What analog I/O is involved?
What resolution? Are results required in engineering units?

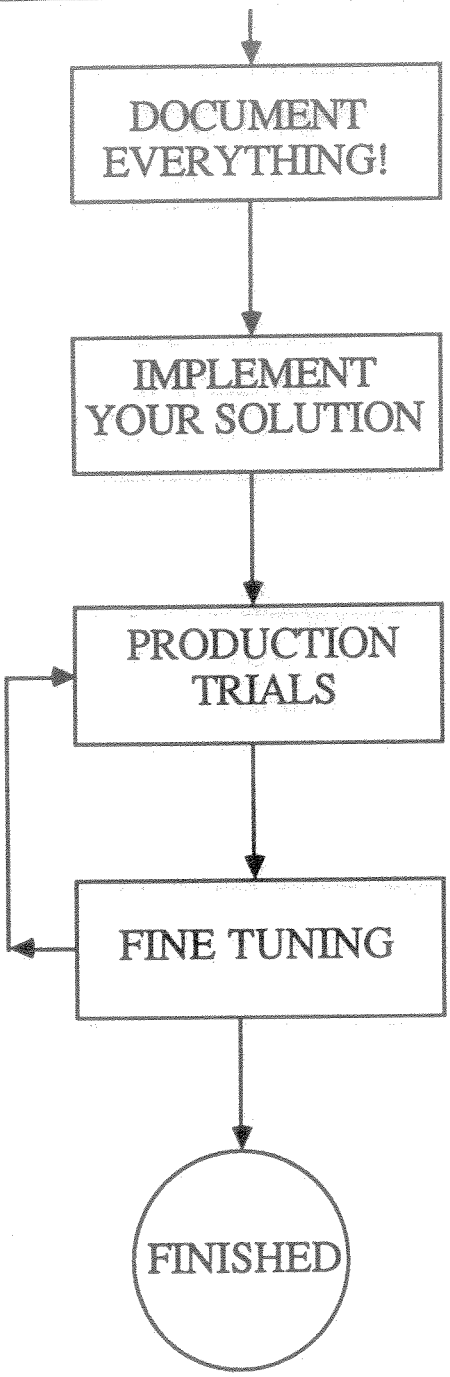
This involves connecting to the actual system and establishing a development environment. Checking voltages and currents to ensure that I/O devices are correctly connected and working is made easier by writing short test words.

Give each ACTION a meaningful name, it helps if each name is readily understood by others who are involved with the system. Warning!! don't get carried away with hiding the underlying hardware!. Keep it simple, just hiding addressess is usually enough.

This is the creative bit. You can test anything instantly! and see immediate results. Try to explore the limits of the system. Always aim for simplicity. (Lateral thinking ability also helps in this phase!) Don't regard the hardware as fixed. Try to replace hardware with software wherever you can.

This is the hard bit, be ruthless in the search for simplicity. Be prepared to back-track as far as needed. (Caution: most people will not appreciate having the problem itself re-defined!)

Ray.Gardiner april 1988



Write the first draft of the program, in the process. Take care to factor the hardware dependant sectons. The higher levels in the software will be self-documenting IF you have exercised care in the choice of names.

Write manuals and operator instructions.

Burn eproms, install hardware, run all cables etc. The things which have changed since you first prototyped the interfaces will now test how well you have factored the hardware.

This stage involves testing exhaustively all of the system. Organize, if possible to TEST changes off line, production seldom appreciates having to work in a start/stop environment.

This final stage of the development cycle should only involve things like changing timing, small details of operator interface, and optimizing small sections of code. Resist the temptation to engage in massive re-writes. (that can be done later!)

Unfortunately, most systems evolve and change over time, rarely will you ever be able to say that's it, all finished.

Usually it happens about a year or two later that some unforseen change is required.

That's when you really find out if you did it right!.

Which leads us neatly into the next topic....maintainability.....

Ray.Gardiner april 1988

MAINTAINABILITY
It has been said that 90% of the world wide programming dollars goes into software maintenance. Briefly, let us explore why this is so.
The main reason software maintenance is so important is that all real world applications exist in a dynamic environment;...change occurs continuously. The type of changes that a real time control system must cope with are:

1...Additional features, and capabilities. These are usually things that weren't required (!) at the time the original program was designed and written.

2...Removal of unwanted capabilities. This may be part of a marketting strategy. Or sometimes if memory space is at a premium, required to make room for new features.

3...Bug fixes, some unintentional features need to be purged.

4...Hardware changes, often an application may be moved to a different hardware environment.

5...The design of the application changes to cope with changes in plant layout. equipment changes.

6...In a manufacturing environment the application may have to cope with product changes, new products may need to be added to the applications capability.

Steps to writing maintainable software.
1. Hide the hardware interfaces. (don't go overboard here, just hide enough to ease readablilty and maintenance)

Use names to describe addresses.
Factor the hardware interface carefully so that changes can be confined.
Purge "magic" numbers from the system.

example:
We wish to read temperatures from a variety of loactions.

```
0 CONSTANT OFFICE
1 CONSTANT AMBIENT
2 CONSTANT PROCESS

: READ-A/D  ( channel number --- a/d-value)
            SELECT-CHANNEL
            START-CONVERSION
            BEGIN COMPLETE? UNTIL
            READ ;

: TEMPERATURE ( channel-number ---- temperature )
            READ-A/D
            CONVERT-TO-DEGREES-C ;
```

We can use this as follows,

PROCESS TEMPERATURE .

We have factored out the parts which will change if (when) the hardware is changed.

Ray.Gardiner april 1988

Does the language help define the solution?

The final software will be a reflection of how well we understood the problem.

By allowing the problem to be probed interactively and viewed from different angles Forth leads to a greater understanding of the problem.

The careful selection of names allows the programmer to "think" about a problem in a high level way.

SIMPLICITY.
A correct solution to a problem is ALWAYS the simplest. This search for simplicity seems to be encouraged by Forth itself. Usually with startling results. Forth itself is a simple solution to a complex problem.

When starting out on a new project it is usual to get mislead by the detail and complexity of the issues to be absorbed. It takes practice and patience to strive for that special view of a problem which leads to a deeper insight and understanding.

:- For more reading on this subject I can highly recommend Leo Brodies "Thinking Forth".

Some real applications

A peach pitting machine which detects split stones and selectively activates pneumatic cylinders to use one of two pitting methods. (networked data collection, macintosh display)

A high speed cherry dispensing system that measures each cherry to correct the count per can for small fragments and ensure accurate count per can. (240 cans/min)

Sawmill control systems with automatic gaging for multiple saw blades.

Plant transplanter control systems which detect when the machine jams and automatically unjam. (multiple cpu's networked for synchronization)

Carton palletiser control systems with multiple pattern selection and video display of pattern .

Boiler control systems, oxygen sensing probe and automatic fuel air ratio trim.

Irrigation control systems.

Automated foundary equipment, gas generators for mould curing systems.

Peach pit detection and sorting systems.

Remote site repeater control systems. Packet radio based distributed control systems.

Research / Laboratory applications.
Underwater data logging system.
Laser pulse control,
Geological microscope data logging.

Current projects; Automated welding monitor, Profile cutter, Peach imaging system, Programmable data logger, Geological survey equipment, LPG gas dispensing system,

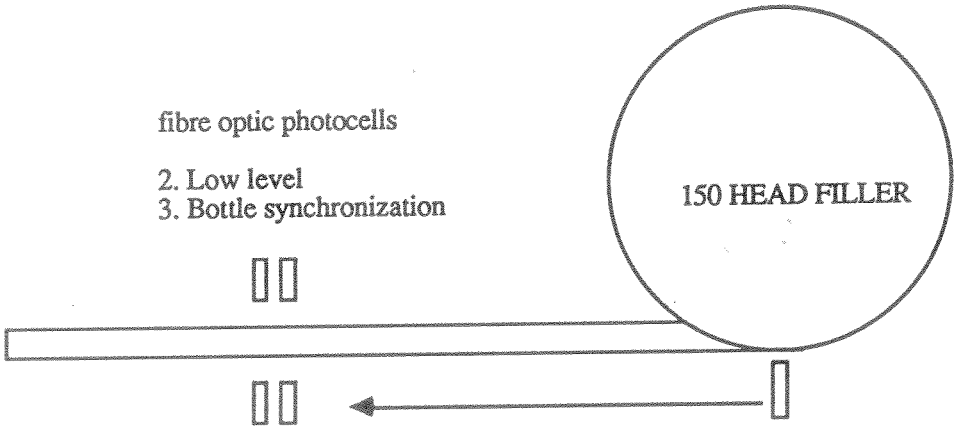
Ray.Gardiner april 1988

Application Example

Product moves at 1000 bottles per minute past photocells which detect underfilled bottles
The underfill is a result of some mechanical problem on the upstream 150 head bottle filler.

We want to automatically stop the line under the following circumstances.

more than X bottles in succession were underfilled
more than Y bottles were underfilled in the last 1000.



For interest, this is a subset of the "real" application. In the real application we also want to identify which head is underfilling. As well as logging the data onto a PC disk for later analysis. The flavour of the application (sic) is not diluted by leaving these finer points out of discussion. (the actual bottle in this instance is sometimes known as a stubbie!)

:

Ray Gardiner april 1988

Application Example

```
VARIABLE BOTTLE-COUNT
VARIABLE BOTTLE-DATA 1000 ALLOT
VARIABLE UNDER                  ( number underfilled in last 1000 )
VARIABLE IN-A-ROW               ( number underfilled in succession )
VARIABLE UNDERFILLED-LIMIT
VARIABLE PER-1000-LIMIT
```

```
( PB is port address for 16 bit sensor interface )
: ?BOTTLE  ( --- flag)   PB @   1 AND ;
: ?UNDER   ( --- 0 or 1) PB @   2 AND ;
: ?RESTART ( --- flag)   PB @   4 AND ;
: STOP-LINE   PB @   8 OR   PB ! ;          ( energize line interlock relay )
: START-LINE PB @   8 -1 XOR AND PB ! ;
: WAIT-FOR-RESTART  BEGIN ?RESTART UNTIL START-LINE ;

: UPDATE-HISTORY ( n --- )
        BOTTLE-COUNT @  BOTTLE-DATA + !
        BOTTLE-COUNT @  1+  1000  MOD BOTTLE-COUNT !

: LOOK          ( --- )
        BEGIN ?BOTTLE UNTIL
        UNDER?  DUP  IN-A-ROW +!  UPDATE-HISTORY ;

: UNDERFILLED?  ( --- f )
        IN-A-ROW @   UNDERFILLED-LIMIT @  > ;

: UNDER-IN-LAST-1000  ( --- n )
        BOTTLE-DATA  DUP 1000 + SWAP DO  I  @ + LOOP  ;

: LAST-1000? ( --- f )
        UNDER-IN-LAST-1000  PER-1000-LIMIT @ > ;
```

```
: SHOW-DATA  ( --- ) ( dump data to terminal for diagnostic )
        CR  BOTTLE-COUNT  ?  IN-A-ROW  ? UNDER-IN-LAST-1000 .  ;

: DECIDE    ( optionally stops the line )
        LOOK
         UNDER-FILLED? LAST-1000?  OR
        IF STOP-LINE  WAIT-FOR-RESTART  THEN ;

: INITIALIZE  ( set up variables )
        0 UNDER !  0 IN-A-ROW !  BOTTLE-DATA 1000 0 FILL
        25  PER-1000-LIMIT !  5  UNDERFILLED-LIMIT ! ;

: BOTTLER  ( this runs the application )
        INITIALIZE  START-LINE
        BEGIN  DECIDE  AGAIN ;
```

Ray.Gardiner april 1988

Applications of Asyst in Digital Signal Processing

by

R. Prandolini, M.P.Moody and J. Miorandi*

ABSTRACT

ASYST is a FORTH based software package which consists of a Mathematics and Graphics Module, a Data Acquisition Module, a Data Analysis Module and a General Purpose Interface Bus (GPIB) Module. The wide range of inherent functions and the flexibility of programming allow sophisticated turn-key signal processing algorithms to be produced in relatively short time.

This paper discusses the use of ASYST in digital signal processing applications and describes one example in the area of speech processing.

1. INTRODUCTION

The applications of Digital Signal Processing (DSP) techniques are becoming increasingly diverse and sophisticated. Researchers in various fields require DSP algorithms to analyse data, while researchers in DSP applications require a system which makes the use of standard DSP algorithms and graphics easy and simple. Asyst is such a system. It allows DSP algorithms to become a tool for researchers, engineers and educators.

2. OVERVIEW OF ASYST

Some of the features of ASYST which make it particularly useful for non-real-time Digital Signal Processing (DSP) are discussed below. Each of the four Modules are addressed with a few examples to illustrate the way in which programs can be written, and to show the ease with which the system can be used.

2.1 Module 1: Mathematics and Graphics

The Base Module [1] contains the FORTH kernel. It can be used in interpretive mode or can be used to define or compile new programs or 'words'. Programs can call previously defined words or inherent words, so it is easy to write programs in modular form. This flexibility is useful when developing DSP algorithms which call functions regularly. Stack based (Reverse Polish) arithmetic is used

* Senior Tutor, Head of School and Tutor respectively in the School of Electrical and Electronic Systems Engineering, Queensland Institute of Technology, Brisbane.

in which any function performed replaces the arguments on the stack. For example, Y X + replaces variables X and Y with their sum. X and Y could have been defined as real, integer or complex variables, vectors or arrays of any dimension. Matrix multiplication of three compatible complex matrices, for example, is as easy as A B C * *. Such functions as MAX, MIN, MEAN are inherent.

Extensive graphics is also provided in the base Module. The word Y.AUTO.PLOT plots an array with scaled X and Y axes, labels, tick marks and grids.

2.2 Module 2: Data Analysis

The Data Analysis Module [2] includes operation on polynomials, vectors, matrices, eigenvalues, eigenvectors and graphs. Features that are of interest to DSP applications include polynomial integration and differentiation, root extraction, determinants, solution of simultaneous equations, least square polynomial fits, data smoothing, peak detection, convolution, filtering, Fast Fourier Transforms and contour plots. The use of some of these features will be discussed further in later Sections.

2.3 Module 3: Data Acquisition

The Data Acquisition Module [3] allows I/O board independent code to be written which addresses a range of I/O boards to be controlled by software. Once configured, Analogue to Digital (A/D), Digital to Analogue (D/A) and Digital I/O are easily implemented. The I/O can operate in the foreground, background or via Direct Memory Access (DMA) control.

2.4 Module 4: General Purpose Interface Bus (GPIB)

The GPIB Module [4] contains drivers for interfacing (via a GPIB board) the host computer to measurement equipment fitted with a GPIB. This allows data to be captured and processed using the full power of ASYST.

3. DIGITAL SIGNAL PROCESSING WITH ASYST

Digital Signal Processing (DSP) is the term given to the processing of discrete data using mathematical techniques. The discrete data are most commonly captured time samples, for example, during a test on a vibration system; however they may take many other forms, such as a 'snapshot' of complex voltages on the elements of an antenna array (the independent variable is spatial), or may be two dimensional, such as pixels on an image taken from a photograph.

Many transformations of data are used in DSP to provide estimates of various parameters. The most common of these is the Fourier Transform, which is an estimate of the

magnitude and phase of spectral components of the signal. Historically, DSP was concerned with seismic, demographic and financial data, and analogue systems simulations since these were acceptably performed in non-real-time. The availability of fast computing devices has meant that DSP has spread into many other fields such as medical engineering, acoustics, sonar, imaging, speech recognition, data communications and control automation [4,5]. Many of these can now be processed in real-time with special purpose processors.

3.1 DSP Algorithms using ASYST

In order to demonstrate the simplicity and flexibility of ASYST for DSP applications, the implementation of some typical algorithms will be discussed.

3.1.1 Waveform Generation

Any mathematically defined waveform may be produced by using the standard mathematics supplied. For example, 100 points of a sine wave may be defined by first producing a scaled ramp of time values, and converting to a sinewave: '100 ramp 100 / 1 - 2 * PI * SIN'.

Complex waveforms such as Pseudo Random Binary Sequences (PRBS) may be generated from their generating polynomials directly, by simply dividing a residue polynomial into unity. Such sequences are used in spread spectrum communications, data encryption, error correction, circuit analysis, system identification, and radar.

3.1.2 Waveform Inspection

After data is captured by the Data Acquisition Module, it may be necessary to look at the data graphically to select points of interest. A 'SCROLL' function allows an array to be moved across the screen, zoomed or contracted, and cursors placed to make measurements. The part of the array selected may then be processed in some of the ways previously mentioned.

3.1.3 Fast Fourier Transform

The Fast Fourier Transform is an efficient algorithm for calculating the spectral (periodic) properties of a sample of data points. In order to optimize the efficiency of the algorithm, the number of points is constrained to be a power of two (i.e. 16,32,64 etc.). The result of the transformation is the spectrum, or the magnitude and phase of the fundamental and harmonic components of a waveform assumed to be a repetitive version of the data points within the observation window. The word 'FFT' will produce the

complex spectrum (i.e. magnitude and phase) of an array of real or complex data points.

3.1.4 Convolution and Correlation

There are several ways in which to compute a convolution or a correlation. Convolution in one domain is equivalent to multiplication in the other. For example, filtering in the frequency domain can be performed by either multiplication by a transfer function in the frequency domain or by convolution with an impulse response in the time domain. Correlation is similar to convolution, except for a reversal in order of one set of data points. Correlation is normally used to determine the degree of similarity of two sets of points.

Convolution and correlation may be periodic or aperiodic, depending on the way in which it is calculated. Aperiodic convolution may be computed by the word 'CONV.APER', while a periodic result may be calculated most efficiently by the algorithm 'ARRAY_A FFT ARRAY_B FFT * IFFT, i.e. the result is the inverse Fourier Transform of the product of the Fourier Transforms.

Periodic correlation is performed by conjugating one of the Transforms before multiplication, i.e. 'ARRAY_A FFT ARRAY_B FFT CONJ * IFFT', while aperiodic correlation may be performed in the same way using arrays padded with zeros.

3.1.5 Homomorphic Processing

Homomorphic processing is commonly used to separate excitation functions from the response to those functions. Two common examples of its use is in echo removal and speech analysis for pitch frequency and vocal tract response. Blind deconvolution can also be performed in order to remove the effects, for example, of room acoustics from recorded signals. In all of these cases, the resultant signal is the convolution of a pulse train (vocal cord vibration, echo epochs), with a response to these pulses (vocal tract response, original signal). Since the waveforms are convolved in the time domain, they are therefore multiplied in the frequency domain. If the logarithm of the spectrum is taken, this product becomes the sum of two logarithmic functions, which can be separated by ordinary linear filtering. An inverse transformation can be performed on either of the separated signals to produce either the train of impulses or the response. An intermediate result in this type of processing is called the CEPSTRUM (SPECTRUM reversed). It is computed by the following: 'FFT LN IFFT', i.e. the inverse Fourier Transform of the Logarithm of the Fourier Transform of the original signal. The CEPSTRUM is a time domain signal.

3.1.6 Digital Filtering

Filtering by any arbitrary filter response can be performed directly from the product in the frequency domain of the signal spectrum by an array of (complex) filter coefficients: 'ARRAY FFT FILTER_COEFF * IFFT'. Various prototype filters based on their analogue equivalents can be designed.

4. A TYPICAL DSP APPLICATION

ASYST is a useful tool since a turn-key system can be generated which may be invoked by single words. In order to demonstrate this, an algorithm has been written which captures a speech signal, allows selection of part of the sample for further processing via the SCROLL and CURSOR functions, calculates the CEPSTRUM and plots spectra, cepstra, vocal tract response, excitation function and a coloured spectrogram. This type of system has been used in the forensic analysis of tape recordings for edit detection and speaker identification. Since ASYST generates resident code, the results are computed rapidly, and at single key strokes if desired.

Some typical results (screen dumps) are displayed below. Reproduction in black and white has disguised some of the effects evident in colour. Figure 1 is the captured time signal of speech using a data acquisition board. Figure 2 is the spectrum of this signal, note that there are several frequency spikes which make up the fundamental pitch frequency and the spectrum of the vocal tract response. Figure 3 is the logarithm of the spectrum. The inverse FFT is the cepstrum, Figure 4. By modifying the cepstrum, (Figure 5), the smoothed log spectrum results (Figure 6). Exponentiation gives the cepstrally smoothed spectrum showing clearly the formant frequencies (Figure 7). The vocal tract response, Figure 8, results from the invers-Fourier transform. The difference between Figure 6 and Figure 3 is the excitation log spectrum, Figure 9. Its anti-logarithm (Figure 10) gives the excitation function, Figure 11.

5. CONCLUSION

ASYST is a useful tool for research, development and general signal analysis tasks. Its large suite of commands, its I/O capability and computation speed allow complex algorithms to be written quickly and efficiently.

It has been found to be very useful in the teaching environment, since the powerful automatic plotting routines allow results to be reviewed as they are produced. The I/O interface allows students to process real signals. Other

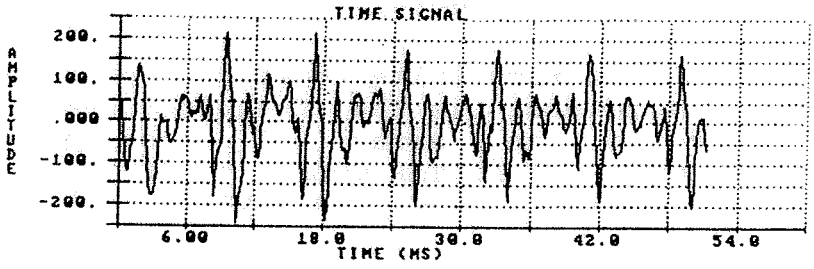


Figure 1.

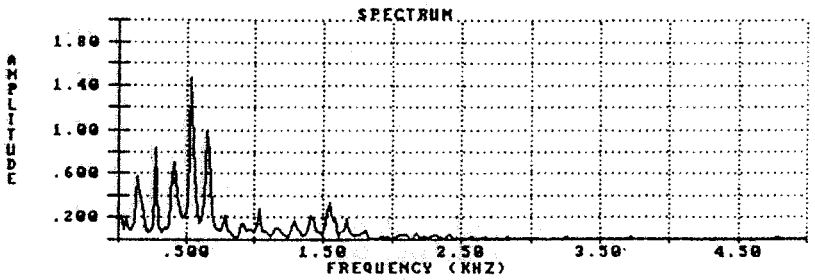


Figure 2.

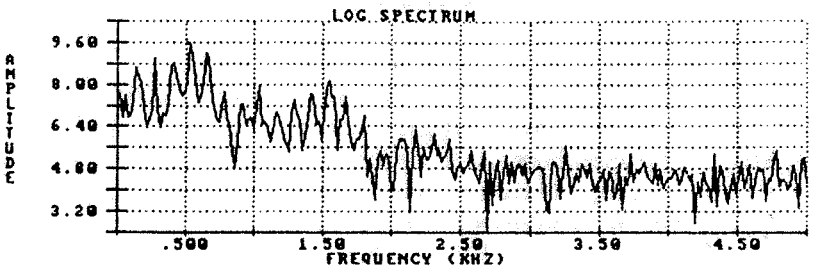


Figure 3.

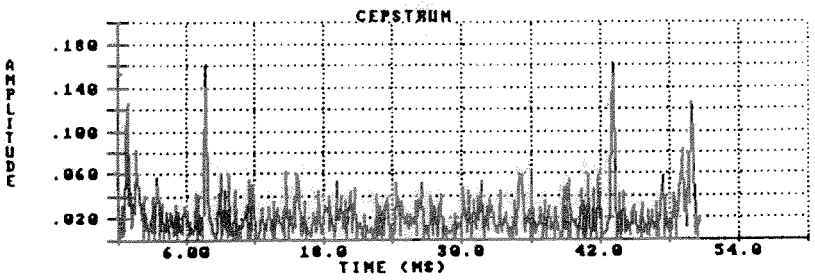


Figure 4.

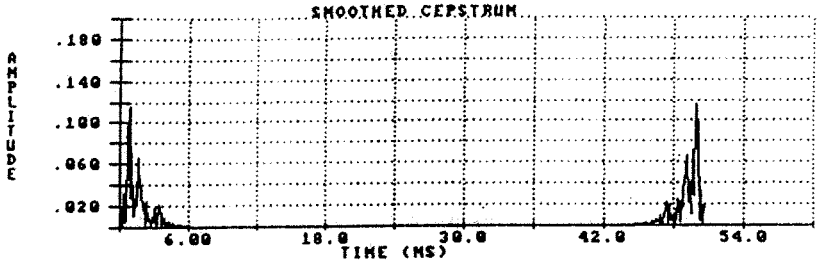


Figure 5.

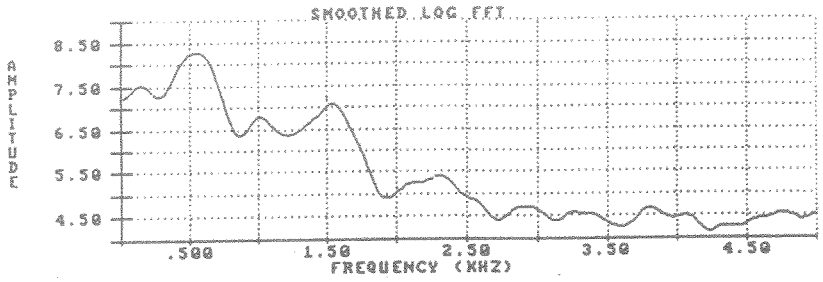


Figure 6.

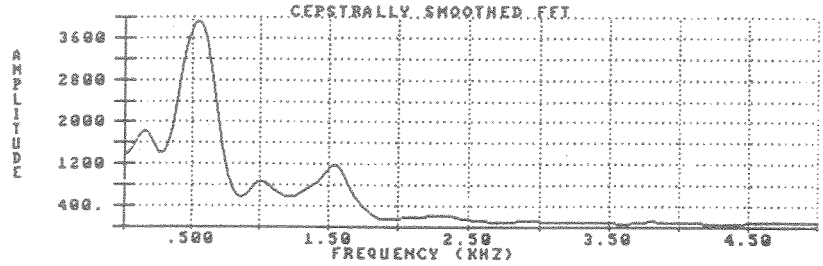


Figure 7.

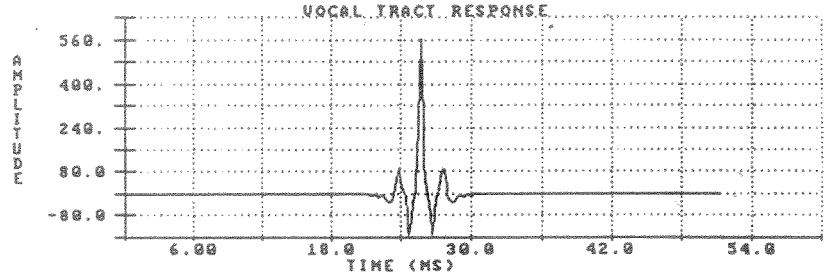


Figure 8.

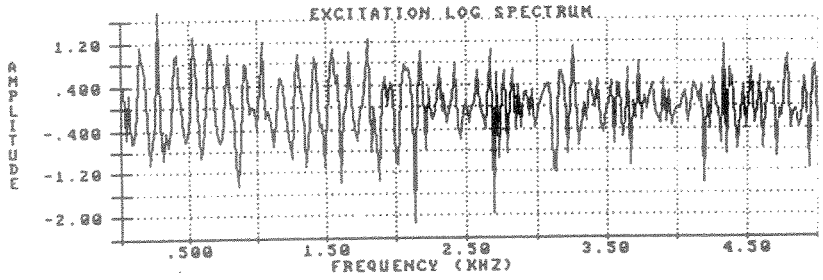


Figure 9.

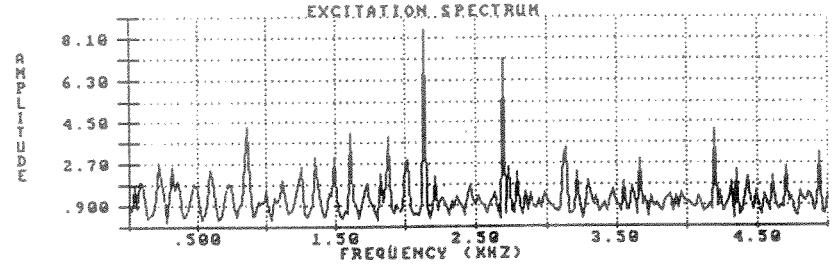


Figure 10.

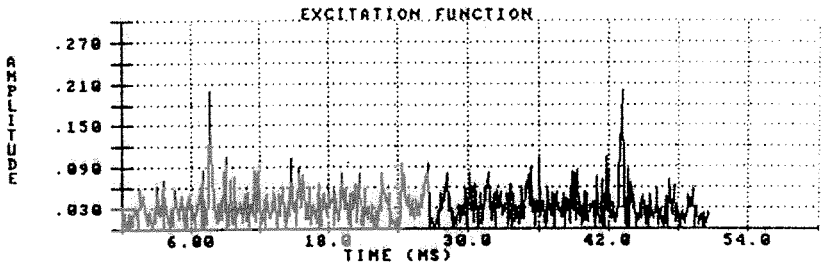


Figure 11.

DSP software packages are also available, such as SIG and ILS, however, the flexibility of ASYST allows greater freedom for the student to try his/her own ideas. ASYST has proven to be an excellent asset for DSP applications.

6. REFERENCES

1 "ASYST Module 1. System, Graphics and Statistics." Macmillan, Inc., 1985, New York.

2 "ASYST Module 2. Analysis." ibid.

3 "ASYST Module 3. Acquisition." ibid.

4 "ASYST Module 4. GPIB." ibid.

5 Rabiner L.R. and Gold B., "Theory and Application of Digital Signal Processing." Prentice-Hall, USA, 1975.

6 Oppenheim A.V. and Schafer R.W., "Digital Signal Processing." Prentice-Hall, USA, 1975.

Forth and Prolog on the Forth Machines

L. L. Odette
Applied Expert Systems, Inc.
One Cambridge Center
Cambridge, Massachusetts 02142
USA

and

W. Wilkinson
1269 Commonwealth Ave, #11
Boston, Massachusetts 02134
USA

Abstract

Our research focus is the problem of engineering expert system technology into systems with severe resource bounds, difficult performance constraints, and non-standard hardware. Good examples are embedded systems for intelligent real-time data acquisition and control. We argue that Forth can profitably be viewed as a platform language for both the real-time and knowledge-base portions of such systems. This paper describes our development work on a Prolog compiler that emits code for an abstract Prolog machine. The Prolog machine is implemented in Forth; versions have been built for the Novix Forth engine and a version is in development for the Harris RISC-based real-time control processor.

Introduction

Commercial development environments with effective support for developing knowledge-based systems are best suited for large and complex applications (PICON, Hawkinson et al., 1985; G2, Wolfe, 1987). Knowledge-based systems of intermediate size, which might be the basis of intelligent real-time controllers, are difficult to move off the high-end machines once developed. There are even fewer delivery paths for smaller knowledge-based systems, which need to be engineered into products and then delivered in quantity.

One of the most difficult tasks in engineering intelligence into real-time applications is balancing the requirements for performance, functionality, and integration. For example, a good approach to achieving high performance and a high degree of integration is to use a single special purpose programming language for both real time processing and representation and reasoning, running on a single high speed processor (eg. Wright et al., 1986). However, these choices may conflict with achieving the necessary functionality (if the representation and reasoning language is too primitive), delivering on time/budget (if the programming tools are limited or if the special processor or its memory is too expensive), or deploying on multiple target machines (if the single special programming language is not portable).

The risk averse will usually trade off functionality and performance for the ability to integrate the application with an existing technology infrastructure. The more agressive will sacrifice integration for functionality and performance because it may enable a solution unattainable otherwise. We believe the price-performance gulf between the two approaches can be bridged by Forth technology.

Approach

Our basic strategy is to maintain a conventional approach to real-time systems where possible. We have therefore focussed on issues of functionality, integration and performance of the representation and reasoning component of the system. To achieve the necessary functionality with a high degree of integration, we use a real-time language (Forth) as a platform on which to implement a standard AI language (Prolog). By choosing the right platform language we are able to maintain high performance.

Threaded languages like Forth meet many of our design criteria for performance and integration. Of particular significance for the implemention of high level languages is the oft-made observation that Forth is well suited to simulating abstract machines. This capability provides portability, since Forth implementations exist for so many machines, and reduces the performance impact of implementing one high level language in another.

The fact that hardware support for subroutine threading is found in several new machines from the RISC school of machine architecture means that the upper bound on performance is very high; ultimately performance of languages like Lisp and Prolog on inexpensive Forth machines may be competitive or even superior to implementations available on high powered and high priced workstations.

Our strategy for achieving the functionality of the representation and reasoning component that is required to build knowledge-based systems is to implement standard AI languages (Lisp, Prolog, OPS5) using the threaded code technique and then build the expert system application in that language. The deciding factors in choosing an AI development language are that the language be well known, and that there is available a body of literature describing solutions to common programming problems in AI. The languages Lisp, OPS5 and Prolog meet these criteria. OPS5 and Prolog also have small kernels so it is relatively easy to implement a complete version of either language,

and both languages may be simulated on abstract machines (Warren, 1983; Forgy, 1982).

Our particular focus is compiling Prolog to Forth. The Prolog implementation language is well thereby suited for developing real-time applications in its own right (and there exits an extensive body literature on Forth solutions) and provides portability and performance. Our previous work has shown that such an implementation strategy can produce code that is compact and fast (figure 1), and the representation and reasoning component is tightly integrated with the real-time component because they share the same run-time mechanisms.

Real-Time Prolog

A multi-pass Prolog compiler has been implemented which ultimately compiles to the Forth language. Early versions of the compiler (Odette, 1987) produced code for a Prolog Virtual Machine (PVM), based on the machine described by Bowen et al. (1983). This PVM is a stack machine (0-address) and while not as sophisticated as the Warren Prolog machine, it is a good fit with Forth (a stack machine), and very portable.

The Prolog compiler described above has been ported to a prototype version of the Novix Forth engine and timed at 6,000 LIPS (Logical Inferences Per Second -- a measure of function calling speed determined with a standard program called "naive reverse") with a clock rate of 4 MHz (Odette and Wilkinson, 1986).

A compiler based on the Warren Abstract Machine (WAM; Warren, 1983) has also been developed; this version runs twice as fast (12KLIPS at 4MHz.). Improvements on the Novix design by Harris Semiconductor have led to a Forth engine designed to run at 15 MHz. (Jones et al., 1987), suggesting that this hardware could support a Prolog at 50,000 LIPS. This is much faster than C language implementations of the Warren machine on the VAX (Gabriel at al., 1985), almost twice as fast as the best Prolog

compiler code on the Intel 80386 class machines, and approaches the speed of compiled Prolog code on Sun workstations.

Our current efforts involve the development of a complete Prolog system for use with the Harris FORCE chip set. The Prolog compiler is written in Prolog and compiles the source code to the instructions for the U.C. Berkeley Programmed Logic Machine (PLM; Fagin and Dobry, 1985). The PLM is a variation on the WAM.

The Prolog compiler runs on an IBM PC and emits PLM instructions into an ASCII text file. The PLM code is then used as input to the FCompiler (Silicon Composers, 1988), a cross-compiler that will compile a subset of Forth for the Harris Force chip set (Forth Optimized Risc Computing Engine). The result of the cross-compilation is an image file that can be uploaded from the PC into a plug-in Force co-processor board. The development system architecture for the Prolog language is illustrated in figure 2.

An example of an application that is a good candidate for this approach is the astronaut interface designed for a series of spacelab experiments (Paloski et al., 1986, 1987). This application is currently based on a full Clocksin and Mellish (1981) Prolog interpreter, implemented in Forth (Odette and Paloski, 1987) and running on an IBM PC clone with interfaces to special purpose hardware for data acquisition, communications and control. The Prolog interpreter provides an interface for executing Forth programs from Prolog . In contrast with the development environment of figure 2, the interpreter approach means that both knowledge base and real-time components are interactive.

The Forth and Prolog code comprise a real-time knowledge based system designed to aid crew members in performing a series of experiments aboard the First International Microgravity Laboratory (IML-1) Spacelab mission. The series of experiments are known collectively as the Microgravity Vestibular Investigation and are designed to study the role of the inner ear in Space Motion Sickness. The astronaut interface helps an astronaut to properly configure the on-board physiological data acquisition and control system. The knowledge based component of the system is designed to reduce the

impact of task complexity and scheduling conflicts on the quality of data collection. In addition, an embedded knowledge base, with information about the mission objectives and strategies for meeting them, may be able to help reduce mistakes in the very likely event that astronauts experience motion sickness and disorientation (indeed, the study is designed to increase the understanding of space sickness).

The IML-1 mission was originally scheduled to fly in May, 1987, but due the Space Shuttle Challenger accident, IML-1 has been postponed until April, 1991. The astronaut interface to the IML-1 application is expected to fly at that time and achieve its mission objective for real-time data acquisition and control with an embedded knowledge base -- all of this in Forth, residing in 64K bytes.

The impact on performance of a Prolog compiler and special Forth hardware would be significant. The knowledge base component (in Prolog) would execute 200-500 times faster, and the Forth code itself would speed up by a factor of 30 to 100. These kinds of performance improvements would enable far more sophisticated applications than are currently possible.

References

D.L. Bowen, L.M. Byrd, and W.F. Clocksin, 1983. "A Portable Prolog Compiler" *Logic Programming Workshop,* Universidade Nova de Lisboa. pp 74-83.

W.F. Clocksin and C.S. Mellish, 1981. "Programming in Prolog," *Springer-Verlag, New York..*

B. Fagin, and T.P. Dobry, 1985. "The Berkeley PLM Instruction Set: An Instruction Set for Prolog" *Technical Report, University of California at Berkely.*

C.L. Forgy, 1982. "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence* , pp. 17-37.

J. Gabriel, T. Lindholm, E.L. Lusk, and R.A. Overbeek, 1985. "A Tutorial on the Warren Abstract Machine for Computational Logic," *Technical Note ANL-84-84,* Argonne National Laboratory.

L. B. Hawkinson, M.E. Levin, C.G Knickerbocker and R.L. Moore, 1985. "A Paradigm for Real Time Inference" *Conference Proceedings, Artificial Intelligence and Advanced Computer Technology* . pp. 51-56.

T. Jones, C. Malinowski, and S. Zepp, 1987. "Standard-Cell CPU Toolkit Crafts Potent Processors," *Electronic Design* , May pp. 93-100.

L.L. Odette, 1987. "Compiling Prolog to Forth," *Journal of Forth Application and Research,* pp. 487-534

L.L. Odette and W. Wilkinson, 1986. "Prolog at 20,000 LIPS on a Novix?," *Proc FORML Conference,* pp. 112-118.

L.L. Odette and W.H. Paloski, 1987. "Use of a Forth Based Prolog for Real-Time Expert Systems II. A Full Prolog Interpreter Embedded in Forth" *Journal of Forth Application and Research,* pp. 477-486

W. Paloski, L.L. Odette, and A. Krever, 1986. "Use of a Forth Based Prolog for Real-Time Expert Systems," *Proceedings Rochester Forth Conference.*

W.H. Paloski, L.L. Odette, A.J. Krever, and A.K. West, 1987. "Use of a Forth Based Prolog for Real-Time Expert Systems I. Spacelab Life Sciences Experiment Application" *Journal of Forth Application and Research,* pp. 487-534

D. H. D. Warren, 1983. "An Abstract Prolog Instruction Set," *Technical Note 309,* SRI Computer Science and Technology Division.

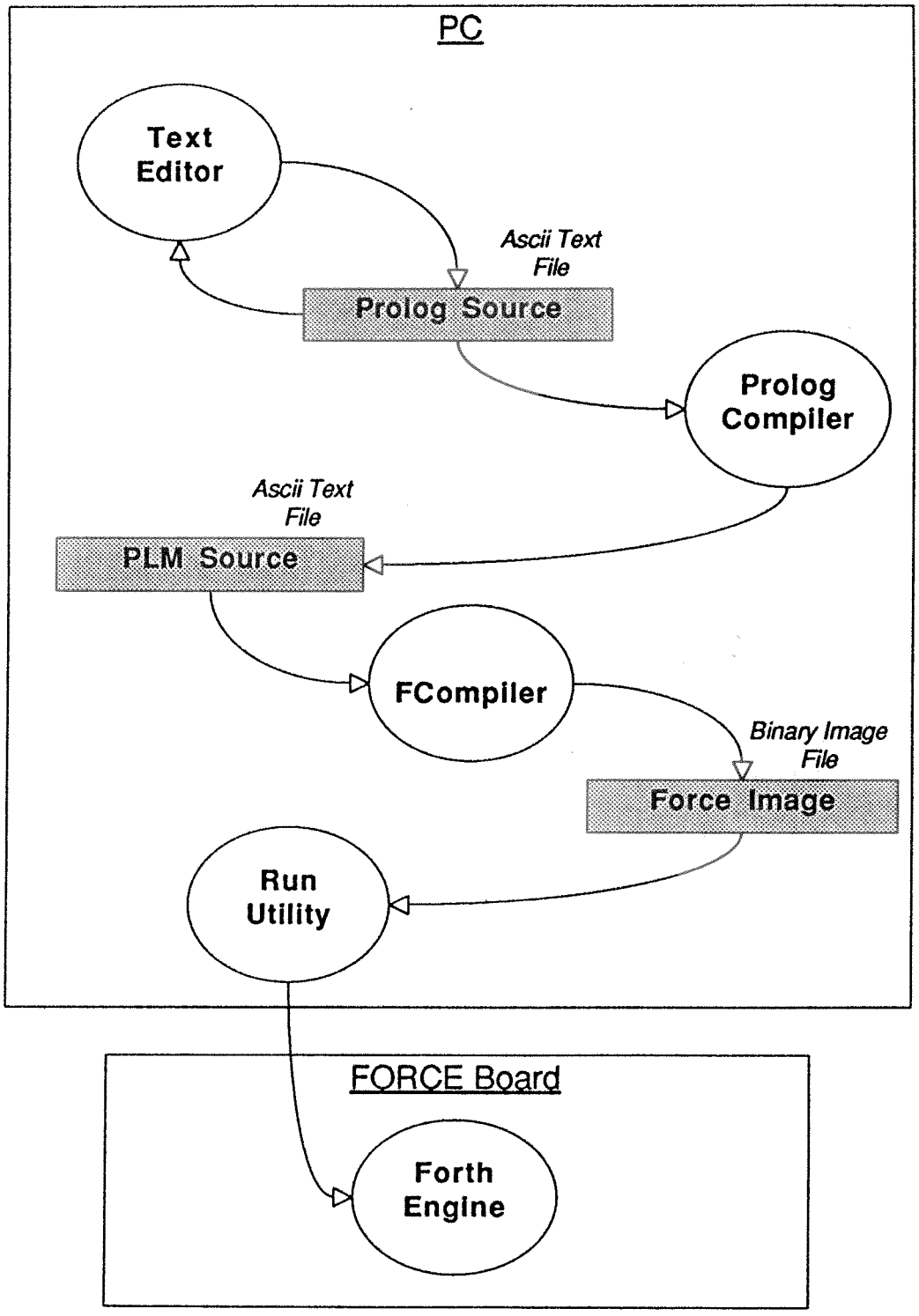
A. Wolfe, 1987. "An Easier Way to Build a Real_Time Expert System," *Electronics,* March, 1987. pp. 71-73

M.L. Wright, M.W. Green, G. Fiegl, P. Cross. 1986. "An Expert System for Real-Time Control," *IEEE Software* (March) 16-24.

Language	Interpreter Encoding Method	Implementation Language	Hardware	Clock (MHz.)	VM Instructions per Second	Reference
Smalltalk	byte code	microcode	Dorado	20	300,000	Deutsch (1982)
Smalltalk	byte code	microcode	68000	12.5	100,000	Deutsch and Schiffman (1984)
Scheme	byte code	macrocode	68000	5	30,000	Schooler and Stamos (1984)
Prolog	threaded code	Forth	NC4000	4	96,000	Odette and Wilkinson (1986)
Prolog	threaded code	macrocode	68000	5	33,000	Pichler (1987)
Prolog	threaded code	Forth	NC4000	4	132,000	Odette and Wilkinson (1987)
Prolog	threaded code	C	VAX 11/780	-	66,000	Gabriel et al. (1985)

Figure 1: Comparison of implementation techniques for AI languages. Each language has an abstract virtual machine that is simulated in software. One useful measure of performance is the rate of execution of the virtual machine instructions. The virtual machine of the first Prolog listed is based on Bowen et. al (1983) and runs at 6,000 LIPS on the NC4000 Forth engine. The other Prologs are based on the Warren Abstract Machine (Warren, 1983). The Warren machine version runs at 12,000 LIPS on the NC4000.

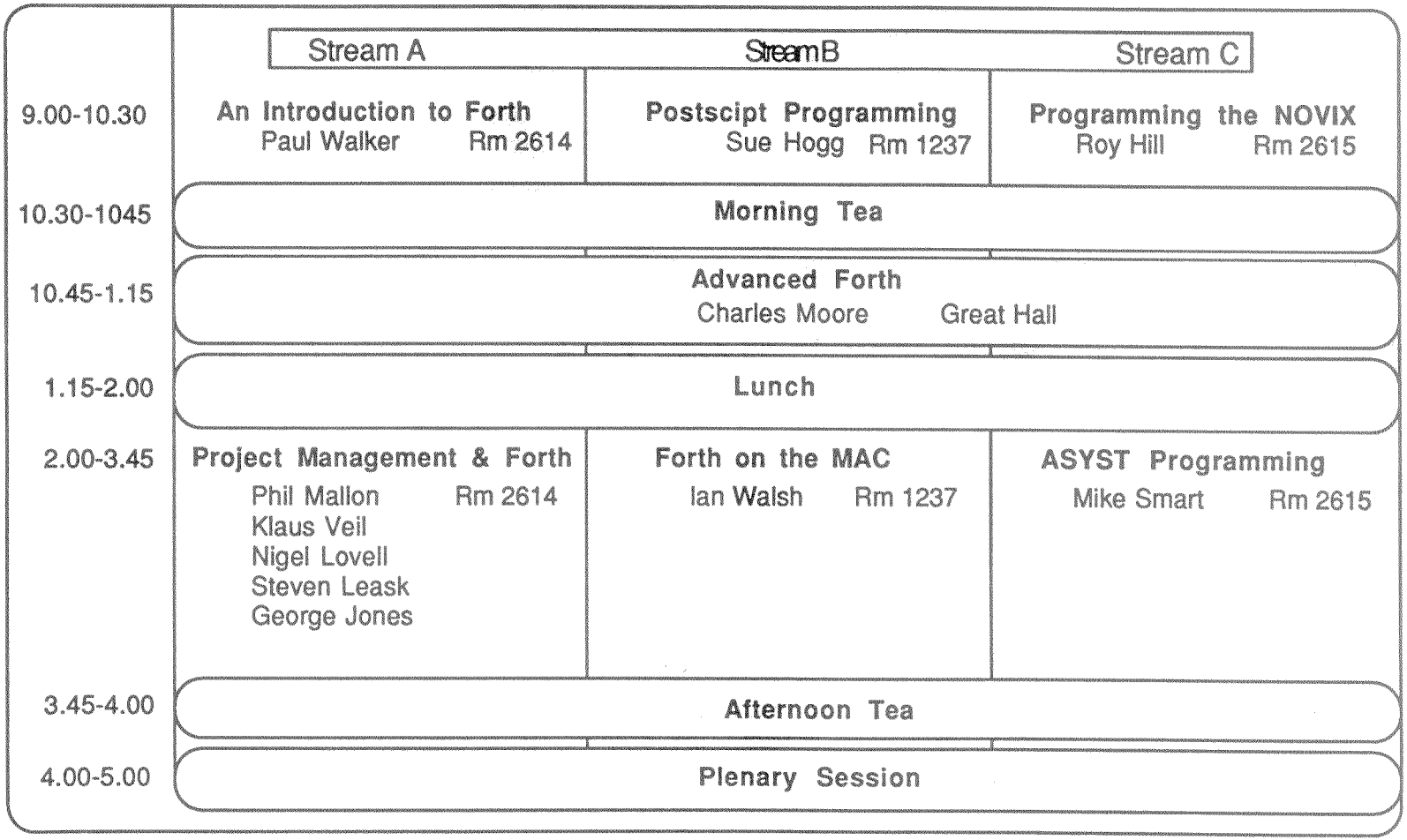
Figure 2: Prolog Language Development System Architecture



Australian Forth Symposium

Forth Workshops

20th May 1988



D56:Forth WS Schedule