AIE

FORTH AND PROGRAMMABLE CONTROLLERS

AIE is the designer and manufacturer of the RTI500 programmable controller. All the software layers which make up the unit have been entirely implemented in FORTH.

What is a Programmable Controller ?

Programmable controllers originated approximately 20 years ago out of the need to provide a more efficient way of implementing large relay logic control systems. Input/Output modules provide the signal conditioning between external devices i.e. limit switches, proximity switches etc., and the internal program. Most large programmable controllers are manufactured using bit slice technology, which execute their own unique limited instruction set like AND, OR and NOT operations on inputs/outputs and/or internal flags.

The requirements of a programmalbe controller may be summarised as follows :-

Hardware constructed for industrial environments, where electrical interference is always present, and environmental conditions are poor. i.e. (high temperatures, high levels of dust and contaminates etc) .

Real time performance, execution of programs in the order of 50 to 100 msec, with faster control sometimes required.

A programming language specific to industrial control, which is easily understood by personnel with no formal programming experience.

Concise documentation of programs to enable quick commissioning , and efficient software maintenance.

Most programmable controllers provide the above features to varying degrees, however the majority of units are suitable for implementing relay logic systems only. Many systems attempt to extend their functions to provide analog processing, however they tend to be very akward and limited.

The RTI500 programmalbe controller steps well past conventional controllers, into the realm of advanced control by providing PID regulators, adaptive control, and feedforward control, with future developments planned for auto-tuning PID regulators.

A software overview of the RTI500 is provided on figure 1.

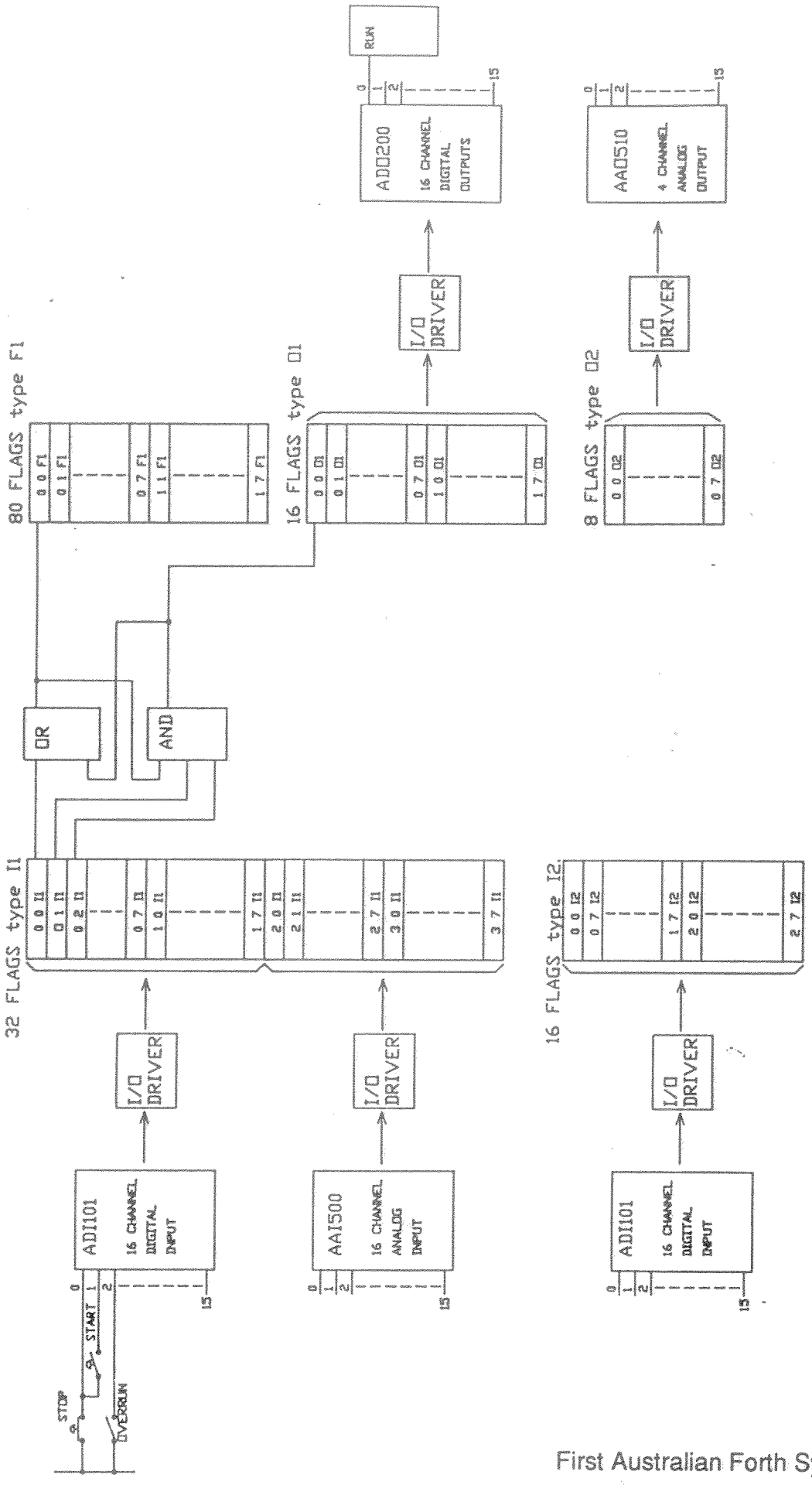

RUN
AD0200
16 CHANNEL
DIGITAL
OUTPUTS
AA0510
4 CHANNEL
ANALOG
OUTPUT
80 FLAGS type F1
16 FLAGS type O1
8 FLAGS type O2
I/O
DRIVER
OR
AND
32 FLAGS type I1
16 FLAGS type I2
ADI101
16 CHANNEL
DIGITAL
INPUT
AAI500
16 CHANNEL
ANALOG
INPUT
START
STOP
LIVE/RUN

FIGURE 1

The software layers which combined together give the RTI500 great flexibility, may be subdivided as follows :-

- Real time multiuser/multitasking operating system (figure 2)

    a) Priority task scheduler for task execution from 10 msec and upward.
    b) Interrupt driven serial I/O used on the lowest priority task level.
    c) Virtually an unlimited number of tasks with fast context switching between tasks.
    d) Full interactive task control from the operator terminal.

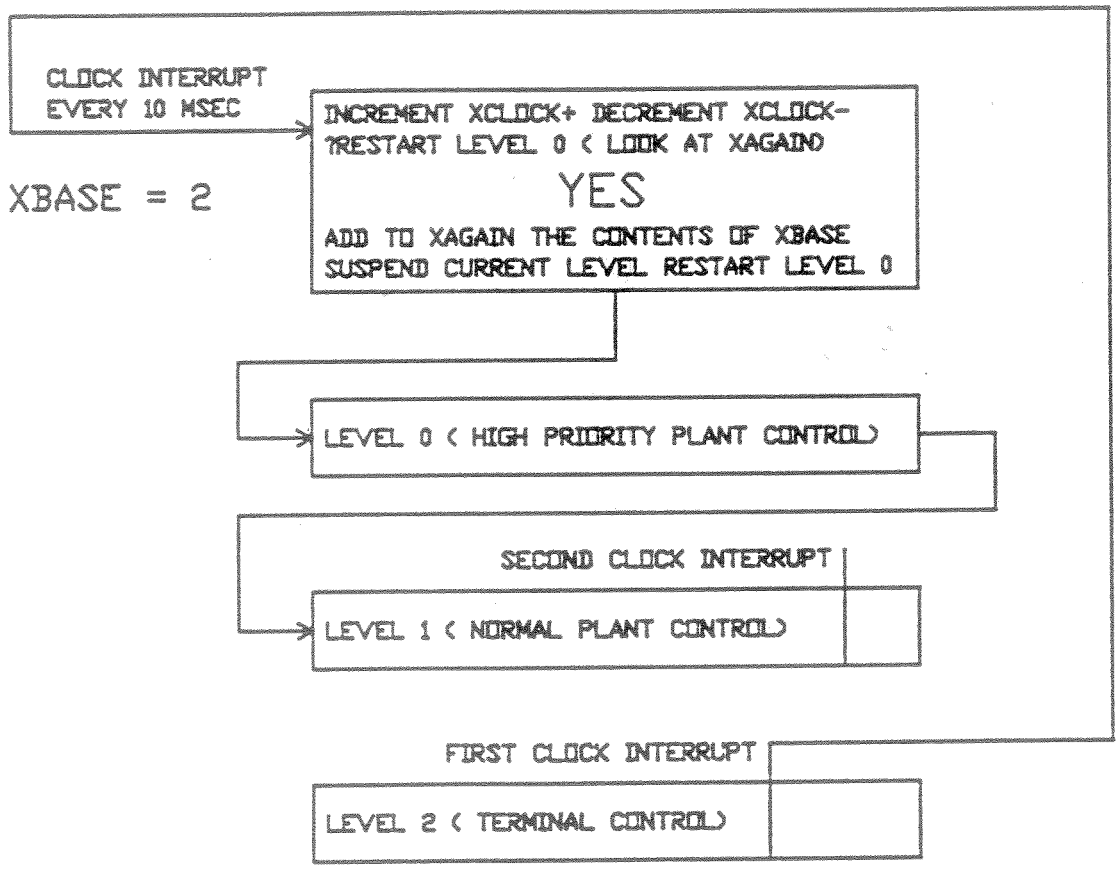


FIGURE 2

- Industrial Forth kernal provides over 800 words in 16 vocabularies.

- Table driven assembler supports the complete 68000 instruction set, with error checking on instruction syntax.

- PC element language for implementing process control systems quickly and efficiently.

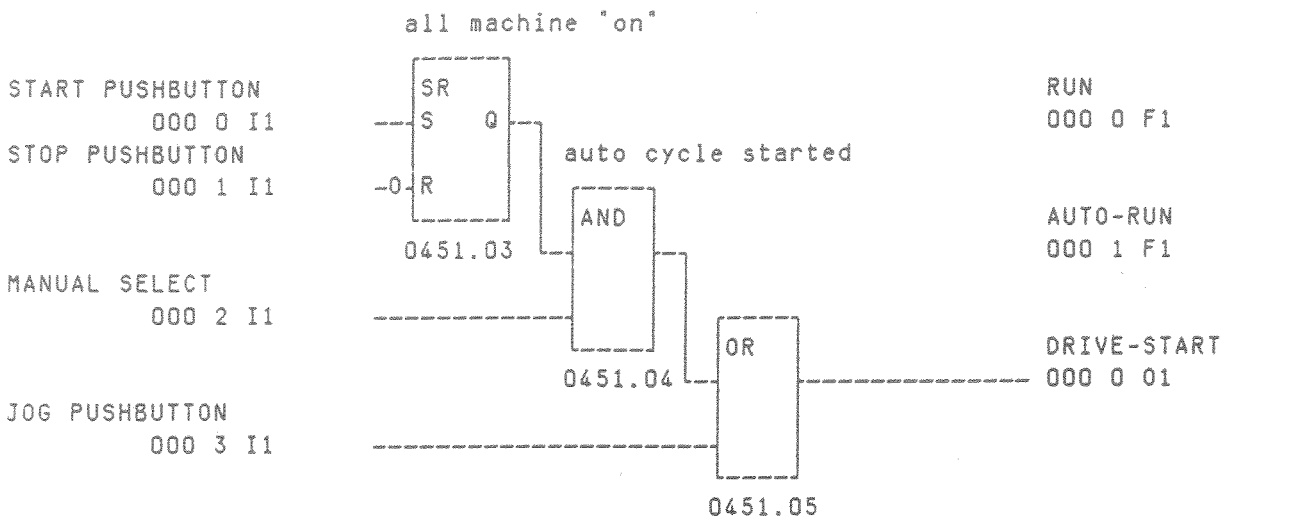
    a) PC element compiler (see figure 3)
    b) PC element documentor for dynamically displaying the application program.
    c) PC element hardcopy documentor for printouts onto a dot matrix printer. (see figure 4)

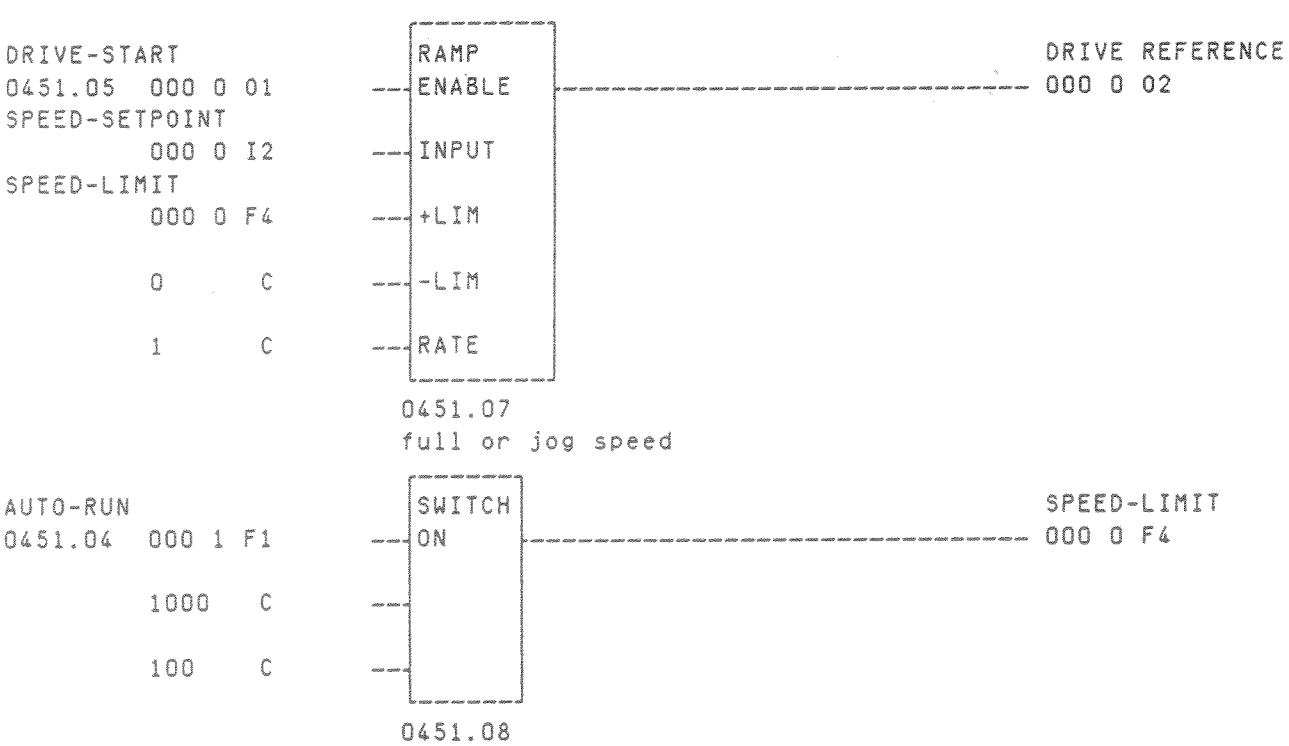
```
451
 0 ( PC elements source code )
 1 PC HOIST-DRIVE
 2 ( START CONTROL AND SPEED REFERENCE )
 3 0 0 I1  // 0 1 I1 ¦ 0 0 F1 SR \ all machine "on"
 4 0 0 F1  0 2 I1 ¦ 0 1 F1 AND \ auto cycle started
 5 0 1 F1  0 3 I1 ¦ 0 0 O1 OR
 6 0 0 O1  0 0 I2  0 0 F4  0 C  1 C ¦
 7 0 0 O2 RAMP  \ control acceleration
 8 0 1 F1  1000 C  100 C ¦ 0 0 F4 SWITCH \ full or jog speed
 9 ENDPC
10 \ LOTS OF COMMENTS may be added
11 \ to describe the reasoning behind this PC element definition.
12 \ We call a PC element definition a PC word.
13 0 TASK HOIST  HOIST-DRIVE ENDTASK
14
15
```

FIGURE 3

SOURCE CODE LISTING ' HOIST-DRIVE 0

START CONTROL AND SPEED REFERENCE






```
 9          ENDPC
10          \  LOTS OF COMMENTS may be added
11          \  to describe the reasoning behind this PC element definition
12          \  We call a PC element definition a PC word.
13          0  TASK  HOIST
            HOIST-DRIVE  ( PC 451.1)
            ENDTASK
```

FIGURE 4

AIE

Additional Utilities -

a) ILAN (industrial local area network) provides communication between one or more RTI500's for DISK services, virtual terminal support, and PC data transfers.
b) Application programs may be promed by attaching a Prom burner to the serial port of the CPU.

The total system just described is entirely PROM resident, and occupies a total of 256 kbytes.

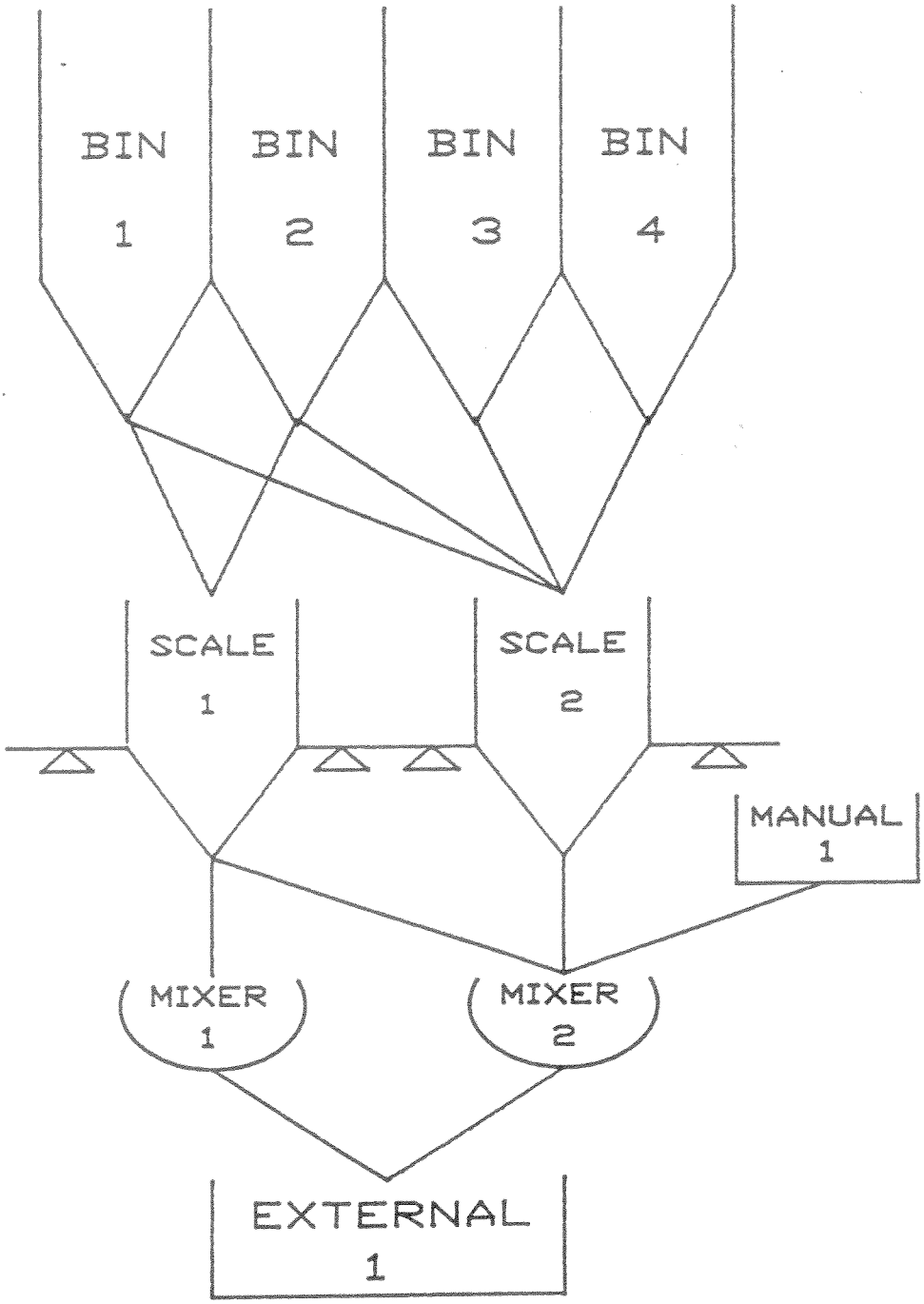
The RTI500 could be described as a hybrid between a conventional programmable controller, and a real time microcomputer. The entire software system has been implemented by Mr. Charles Esson of AIE over the past 4 years. Before joining AIE, Mr. Esson had considerable experience in implementing turnkey control systems based on programmable controllers which were hardwired to micro-computer systems. The microcomputers were programmed in macro assembler , whilst the programmable controllers were programmed in boolean logic. It was obvious that combining the two functionally different pieces of control equipment into one unit would produce quite a powerful controller. Although never having used Forth before, it was evident to us that Forth was the obvious language to use for this purpose.

Just to give an insight into the systems extensibility (inherent in FORTH), twelve months ago we employed a graduate engineer Mr. Ivan Kramersh who's task it was to add a flexible batch control package called "auto batch" to the RTI500. Never having used FORTH before he completed 90 % of the software in 6 months, and has just returned from commissioning our first "auto batch" controlled pelletizing plant in Western Australia. The system ran exceptionally well, and we hope to achieve considerable local and international sales of the RTI500 batching system as a result.

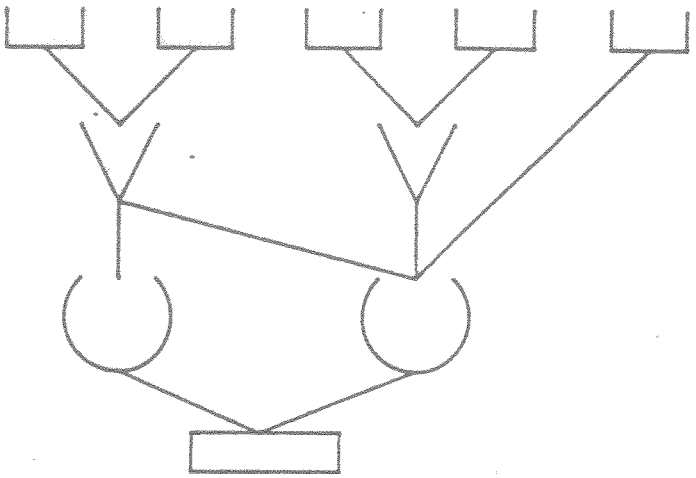
AIE

What is a batching system ?

A batching system is one which combines a number of ingredients defined by a specific formula to produce the end product. Different formulae produce different products from the same basic ingredients. i.e. (oatmeal biscuits as apposed to muesli biscuits). The RTI500 batching system is PROM resident, and is a an additional layer on top of the standard system. An example of a batching system, and its configuration is presented below -



The above system may be functionally decomposed into our basic system elements as follows:



INPUT BINS AND MANUAL STATIONS

FEEDERS

SCALES

FEEDERS

MIXERS

FEEDERS

EXTERNAL EQUIPMENT

In order to configure "Auto-Batch" to the above system, the following code must be executed:

```
1 BIN BIN1 0 0 F1 EMP 0 0 M4 SL
2 BIN BIN2 0 1 F1 EMP 0 1 M4 SL
3 BIN BIN3 0 2 F1 EMP 0 2 M4 SL
4 BIN BIN4 0 3 F1 EMP 0 3 M4 SL

5 MAN MANUAL1 0 4 F1 RTD 0 5 F1 OTD
              0 0 I1 BTN 1 3 01 LMP
              0 6 F1 BUSY 0 7 F1 OKTP
1 SCALE SCALE1
1 0 F1 RTD 1 1 F1 OTD 0 0 12 CRD
1 2 F1 BUSY 1 3 F1 OKTP 0 0 M2 TARE
2 SCALE SCALE2
1 4 F1 RTD 1 5 F1 OTD 0 1 I2 CRD
1 6 F1 BUSY 1 7 F1 OKTP 0 1 M2 TARE

BIN1 SCALE1 1 PATH PATH1-1
0 3 01 FFEED 0 5 01 SFEED 2 0 F1 FF
BIN2 SCALE1 2 PATH PATH2-1
0 5 01 FFEED 0 5 01 SFEED 2 1 F1 FF
BIN1 SCALE2 3 PATH PATH1-2
0 2 01 FFEED 0 2 01 SFEED 2 2 F1 FF
BIN2 SCALE2 4 PATH PATH2-2
0 4 01 FFEED 0 4 01 SFEED 2 3 F1 FF
BIN3 SCALE2 5 PATH PATH3-2
0 1 01 FFEED 0 1 01 SFEED 2 4 F1 FF
BIN4 SCALE2 6 PATH PATH4-2
0 0 01 FFEED 0 0 01 SFEED 2 5 F1 FF
```

```
1 MIXER MIXER1
        3 0 F1 RTD 3 1 F1 OTD
        2 0 01 PWR 3 2 F1 DSA
        3 3 F1 ABRT 3 4 F1 BUSY

        2 MIXER MIXER2
        4 0 F1 RTD 4 1 F1 OTD
        1 2 01 PWR 4 2 F1 DSA
        4 3 F1 ABRT 4 4 F1  BUSY
        4 5 F1 OKTP

        SCALE1 MIXER1 7 PATH PATH7
        1 4 01 FFEED 1 4 01 SFEED 5 1 F1 FF
        SCALE1 MIXER2 8 PATH PATH8
        SCALE2 MIXER2 9 PATH PATH9
        1 0 01 FFEED 1 0 01 SFEED 5 3 F1 FF
        MANUAL1 MIXER2 10 PATH PATH10
        0 7 01 FFEED 0 7 01 SFEED 5 4 F1 FF

        1 EXTERNAL EXTERNAL1

        MIXER1 EXTERNAL1 11 PATH PATH11
        1 7 01 FFEED 1 7 01 SFEED 5 6 F1 FF
        MIXER2 EXTERNAL1 12 PATH PATH12
        1 5 01 FFEED 1 5 01 SFEED 5 7 F1 FF
```

NOTE: All PC database points particular to a system element follow directly after the elements definition and are all defined before the next element definition.

Conclusion

It is very satisfying to be able to say in hindsight, that using Forth was an excellent decision. Taking into consideration a time contraint of 4 man years, and a 256 kbyte limit on PROM space, (512 kbytes with the batching sytem) it would have been impossible for us to implement the RTI500 software in any other language.
Extensions to the RTI500 like the batching system, means that future developments will widen our application base, and perhaps help to give Forth some of the credit it deserves.

Author : Gary Brown,
Systems Engineer.

Australian Industrial Electronics Pty. Ltd.,
47 Gatwick Rd.,
Nth Bayswater,
Vic. 3153

Phone : 03 720 2511.

DESIGN PHILOSOPHY OF AN NC4Ø16-BASED MICROCOMPUTER

by Roy Hill

A little historical perspective may help to understand why a very small (3 person) Australian electronics organisation would be able to design, produce and market an IBM type co-processor board that even the Americans would be proud to claim as their own.

I was introduced to micros at a very early stage in their development, by way of an evaluation board based on a 65Ø2 processor. Those were the days when micros were first introduced to a "dazzled" industry with small, limited power boards which enabled design engineers and OEM's to get their hands on a workable development tool. My "tool" was a SYM-1, made by the now defunct Synertek Systems Inc. This board was introduced to me by a colleague at the firm I had just left, to take up teaching for TAFE. He also introduced me to the exceedingly limited software available for the board - a BASIC interpreter, a Resident Assembler/Editor and Forth. I had some previous experience with BASIC (on a Data General machine) and with assembly, but this "Forth" thing was a totally new language. And Boy, did it take some major re-alignments of my programming paradigms. Prior to this I also had use of a Hewlett-Packard programmable calculator and I wasn't overly impressed with stacks. After all, the HP was a continual source of hardware problems, so that anything that even resembled HP styles was enough to immediately ring alarm bells.

However, a software company in Canada was offering some utilities and a newsletter on a bi-monthly basis, so I rolled up with my subs and "started Forth." It took several agonising weeks become a convert to this terse and compact new language. Anyone who is introduced to this language either becomes a convert or protagonist. I became a zealous covert, anxious to spread the word. I attended a symposium at Queensland Institute of Technology and delivered a paper on a Forth package for the Apple II. Paul Walker attended that meeting and went away also inspired. That inspiration is largely responsible for this Symposium. At this time, I also had cause to have my Lowrey organ serviced by two local service technicians. The problem was rather ticklish and required several return visits. On one of these visits the two technicians noticed my SYM-1 and the fact that it was running Forth. Their interest was immediately aroused and the two technicians (now known as Maestro Pty Ltd) went away, firmly convinced that Forth was a good thing. The next step was to actually use it in software development for several projects and then to adapt that experience to experiments with Forth on the IBM and finally, on a prototype Novix board from Computer Cowboys. From this prototype came Chris and Dan's idea of a total Forth system, based on the NC4Ø16 and capable of being used either inside an IBM, outside any computer with a serial port (including an IBM) and a small 5 volt power supply, or with a simple serial terminal and monitor.

The two main motivating features for the whole project were the existence of Forth and the existence of the Novix to

run it on. As most people are well aware, Forth uses a technique called "Subroutine Threaded Code" to perform its task. In essence this means that Forth programs spend a large amount of their time entering and exiting subroutines. Charles Moore optimised the Novix to enter and exit subroutines in as few as 1 clock cycle - a fairly impressive achievement when compared with existing Von Neumann style processors. Maestro (Chris and Dan) decided that it would be a challenge to produce a board based on the Novix chip. Further discussions produced the idea that a co- processor board for the IBM would offer a large software development advantage. Current Novix boards (including the development system produced by Novix themselves) were far too expensive to enable the average hobbyist to respond to the new technology.

The three of us then thrashed out a specification for the board that we hoped would be fairly impressive. The specification stated:-

(a) The board had to cost well below A$1000 so that the "average hobbyist" could afford it.
(b) The board had to be usable by both existing Forth programmers and complete tyros.
(c) The board had to produce some immediate demonstration of the power of the Novix combined with Forth.
(d) The board had to operate in as wide a variety of combinations as possible:-
    - Inside an IBM XT/AT (not a "friendly", electronically speaking, environment)
    - From the RS232 port of virtually any existing computer
    - From the serial port of a simple Terminal/Monitor

Given this third point, some form of source code storage had to be provided to enable applications to be stored and retrieved after power down. This took the form of providing an on-board EPROM Programmer.

(e) The board had to provide more capability than being "just another evaluation board."

Chris and Dan were prepared to produce the board for virtually cost price, in order to cover the first point mentioned above. Providing slow (but cheap) static RAM as the memory for the board was one way of trimming costs. However, users also had to be able to upgrade to faster chips (and hence, to higher throughput speeds), without major hardware modifications to the board. This was provided for with a programmable system clock and the ability to replace the crystal with a faster version. This is discussed in detail further in the text. Board layout was also a critical factor - long tracks mean high inter track capacitance and lower speeds. Also, design of hardware address decoding prohibited the use of too many gates, as the decoding delays would prove unacceptable. The overall design took two months from initial concepts to first prototype. Incidentally, the prototype worked first time - an extremely satisfactory result, given the complexity of the board. One of the first things that

designers learn about prototypes is "if they work - don't change them." Did we? Yes, I'm afraid we did. Chris has long been interested in speech synthesis and it was decided to add an 8 bit A/D-D/A converter to the board and a small amplifier circuit to produce speech. High quality speech, too. Not Donald Duck with a mouth full of breadcrumbs type speech, but recognizable as coming from an individual. One of the first trials was to record me saying "Welcome to the Maestro Novix board." It's an uncanny feeling to hear an exact reproduction of one's own voice, with no tape recorder in evidence.

A PROGRAMMABLE SYSTEM CLOCK

The 4 x 28 pin chips in the top right hand corner of the board are the RAM chips. The manner in which these chips are used is quite an interesting exercise in computer design. These chips are 43256 type static RAMS. Whilst this paper is not intended to be a treatise on computer design or advanced electronics, it is helpful to understand how this board has been to developed with two main criteria in mind:-

1The board had to run at the highest economically feasible speed, in order to make the most use of the high speed processor.

2The board had to use readily available components that were not too costly to enable the average home enthusiast to purchase.

For these reasons, it was decided to provide the kit with 43256-120 RAM. These chips run at 120 ns and are relatively cheap, but unfortunately, only allow the Novix to run at 4 MIPS. For an extra $70, 43256-100 chips (running at 100 ns) will enable the Novix to run at 6 MIPS. The manner in which Chris has designed this board enables users to make their own cost/performance selection without requiring major board modifications. Let me explain how this is going to be done. The clocks of most computers are designed as shown below in Figure 1.

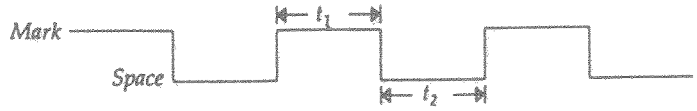
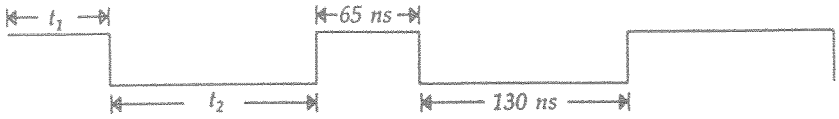


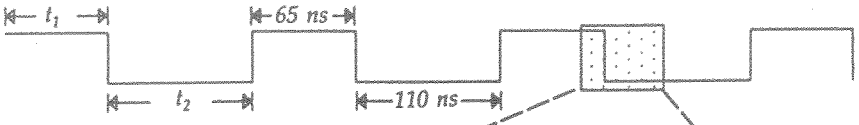
Figure 1. The design of the 'symmetrical' clock found in most computers — t_2 is usually (but not always) the same time as t_1. If t_2 is longer than t_1, the clock is 'asymmetrical'. This is as far as most computers go, in order to allow for RAM (which uses t_2) that is slower than the processor being used.

"t2" is usually (but not always) the same time as "t1". This is known as a "symmetrical clock." If "t2" is longer than "t1", the clock is known as an "asymmetrical clock." This is as far as most computers go, in order to allow for RAM (which uses "t2") that is slower than the processor being used. Let's have a look at how the Novix would work under a symmetrical clock with the chip running at 6 MIPS. To run at 6 MIPS, the Novix would a "t1" of about 75 ns. To provide RAM

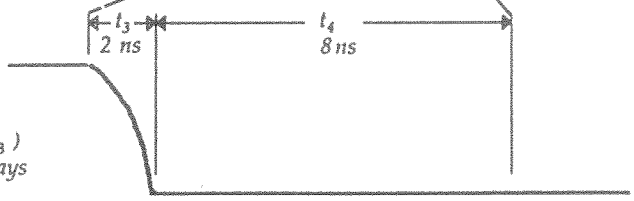
having a similar speed would cost the earth (about $40 PER CHIP and you'd need four times as many of them). This would place the cost of the board way beyond the purchasing ability of all but the most dedicated. However, if we provide the board with an asymmetrical clock, we can make use of slower (read less expensive) RAM.



An asymmetrical clock suitable for use with 100 ns RAM chips.



Asymmetrical clock suitable for use with 120 ns RAM chips.



The effect of 'fall-time' (t_3) and hardware decode delays (t_4) on access time.

Figure 2. Using an asymmetrical system clock ($t_1 * \neq t_2$) allows the use of slower and less expensive RAM.

Note that each of the RAM access cycles is 10 ns longer than the actual designated speed of the chips. This is to allow the hardware time to decode the actual address of the chip being "talked to" and also to allow for the fact that none of the verticals on the diagram is truly vertical. It is more like that shown in Figure 3c. If we also make the clock totally programmable, with variable mark/space ratios, we can make provision for almost any speed of RAM and introduce another very interesting aspect of Novix Power. This is what has been done on the Novix board and its use is very much like the accelerator/brake combination on a car. Normally, the Novix runs at the full speed of the clock, but by placing a value on the top of the stack (which is done by just typing in the number and executing the word CLOCK, the clock speed is changed to the value that was placed on the top of the stack. In practical terms, this means that the board can be user programmed to run at anywhere from full speed to about 2 seconds per clock cycle. The advantages of being able to do this may not be readily apparent, however, this means that:-

1 We can perform dynamic de-bugging of source code, simply by slowing down the clock to its slowest speed and examining the source code as it executes.

2 There are numerous advantages in having the processor operate slowly - educational, demonstrations data and return stack examination

etc., where the on-looker can see the instructions being performed in real time.

A/D AND D/A INTERFACE

The addition of the A/D-D/A converter and associated support circuitry on-board also means that it is possible to develop applications for data acquisition/ control and robotics. The possibilities of high speed video capture and associated applications are within the grasp of the hobbyist.

The final design consideration was that the whole board had to be capable of being built as kit - once again to keep the cost factor down.

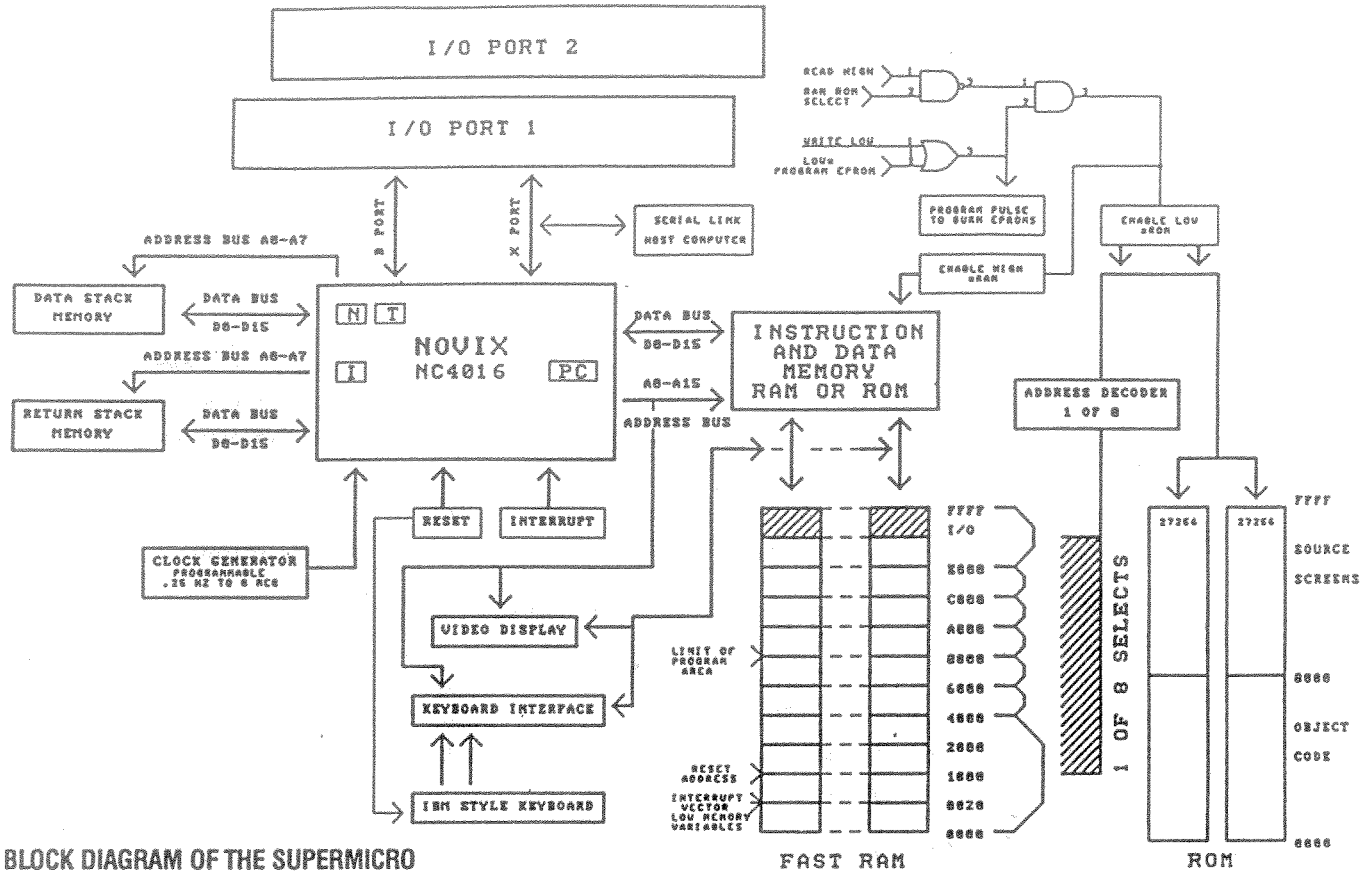
BOARD LAYOUT

The board is designed to occupy one of the 8 bit slots in an IBM PC. However, the provision of a serial port that handles a standard RS232 interface means that it can also be used in an external mode.

The Novix is the square PGA chip (IC1) in the top Left Hand Corner. To its immediate left are the two Data Stack chips (High and Low bytes - IC's 13 and 14). On its right are the two Parameter Stack chips (IC's 11 and 12). The main memory chips are the four RAM chips below the Novix (IC's 29 - 33 inclusive), whilst the system ROM (two x 27512 EPROMS) flanks the main memory (IC's 34 and 35).

The A/D and D/A section is located on the top Right Hand side of the board, and uses the Analog Devices AD7569 chip. This is a fairly fast 8 bit A/D-D/A converter and its I/O lines are also taken out to the DB25 connector on the extreme RHS of the board.

All that remains is to demonstrate its versatility and wait for users to start feeding back the applications to which they are putting the Maestro SuperComputer.



BLOCK DIAGRAM OF THE SUPERMICRO

FIGURE 3 - BLOCK DIAGRAM OF THE SUPERCOMPUTER

PROTOCOL TESTING USING FORTH

S.A. Leask and R.A. McNaughton
Telecom Research Laboratories
Clayton, Victoria.

1 Introduction

A communications protocol is a set of rules which governs the communication between two or more entities such as computers, telephone exchanges, etc. A protocol may be simple or complex, depending on the requirements of the communication. A protocol's complexity may be gauged by how elaborate these rules are, and by how much information is transferred between the entities to perform a given task, e.g. to set up a connection. The more complex protocols require sophisticated tools to verify their correct operation and analyze their performance.

In December 1986, Telecom Research Laboratories and Siemens AG (Germany) embarked on a joint project to develop a test system for the Integrated Services Digital Network (ISDN) Access protocol. Currently, a second project is underway to develop a test system for the Common Channel Signalling number 7 (CCS7) protocol, which is used for the signalling between telephone exchanges. Both these systems are based on the Siemens K1195 Protocol Tester using the FORTH programming environment.

The FORTH language is particularly suited to this type of application. The high level nature and extensibility of the language allow for quick prototyping. In addition, the FORTH interpreter gives the programmer considerable power in interactive testing of his application.

This paper describes the various ways in which FORTH has been used in these projects, and examines its strengths and weaknesses. The programming environment under which the test software was developed is also discussed.

2 What should a protocol tester do

The primary function of any protocol tester is to monitor the data passing between two communicating entities (Figure 1a). In this role, the tester is a passive device, taking no active part in the communication. However, it has access to all the data being transferred, and uses this to perform the following functions.

Monitoring: The data exchanged between the two devices under test (DUTs) is displayed in a meaningful way. A range of display formats should be available, ranging from a primitive hex dump of the data to a comprehensive mode where the value of every bitfield is reported, along with a textual description of the value. Filters should also be provided, allowing only certain events to be reported. In this way, the tester can reduce the amount of output provided so that only the events of interest are displayed. Triggers, which allow

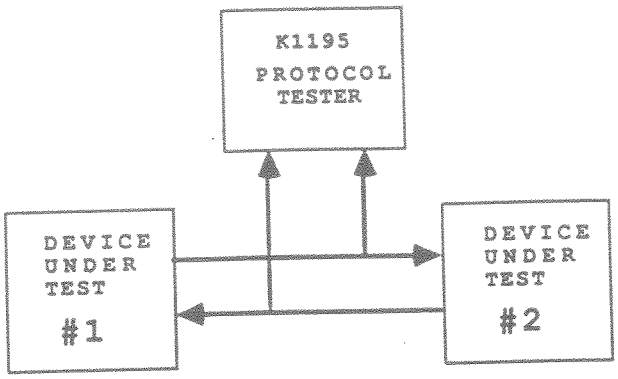


Figure 1a. Using the protocol tester as a monitor.

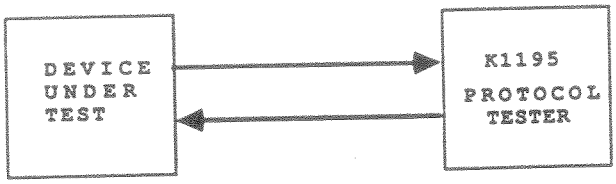


Figure 1b. Using the protocol tester as a simulator.

a user-programmable action to be performed when a particular event occurs, may also be provided.

Logging: In addition to on-line display, a protocol tester should provide facilities to record the data onto a permanent medium such as magnetic tape or disk. This permits the storage of larger amounts of data than can be stored in memory, as well as allowing the data to be analysed off-line at a later stage.

Analysis: In both on-line and off-line modes, the tester should be capable of performing some level of analysis on the received data. This analysis may consist of generating some standard performance criteria, e.g. data throughput, or some user-defined criteria e.g. the proportion of attempted call setups which were successful.

For more comprehensive testing, the protocol tester may take the place of one of the sides of the communication link (Figure 1b). In this mode, the tester takes an active role in the communication, subjecting the device under test to a known set of stimuli, and then checking that it generates the correct response. This mode of operation is known as simulation or emulation, the distinction being that a simulation performs a user-programmed protocol behaviour (possibly incorrect, to check for correct error handling), while an emulation performs a correct, hard-coded protocol behaviour.

In the simulation and emulation modes, all the facilities available in monitor mode remain available. In fact, some of the analysis functions may be used to test the performance of the tester in these modes!

Finally, a protocol tester should provide sufficient support for the user to program a simulation. This support consists, in the case of a FORTH environment, of words to

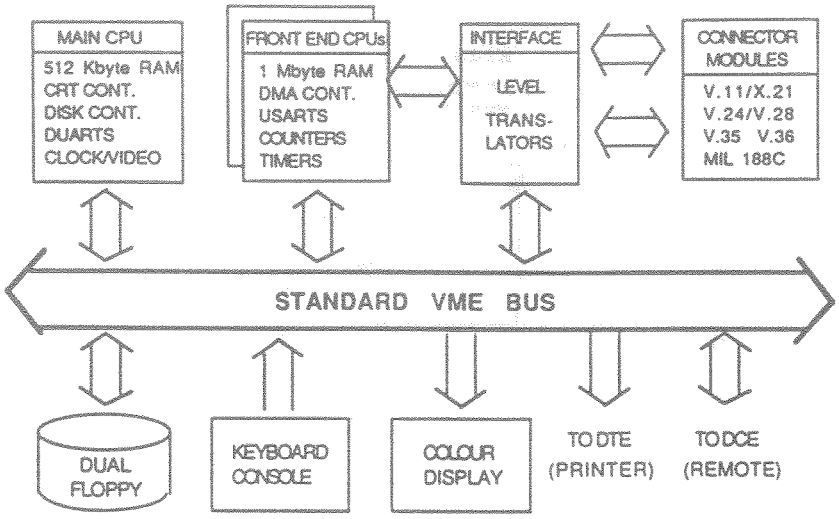


Figure 2. Hardware configuration of the K1195

- Recognize particular events, e.g. to detect the receipt of a particular frame.
- Perform data transmission e.g. to send a particular frame.
- Construct frames prior to transmission in a user-friendly way (as distinct from requiring that the data be specified as a pre-coded string)
- start, stop, and recognize the expiry of timers.

3 Hardware Configuration

The Siemens K1195 Protocol Tester is a multi-processor VME bus based data test computer, which uses a multi-tasking real time software environment (Figure 2).

Motorola 68000 microprocessors are used on each CPU board (running at 8 MHz), each of which maintains its own FORTH dictionary. This allows the simultaneous execution of a FORTH application on each CPU.

The main CPU board contains 512 Kbyte of on-board RAM, and performs such tasks as CRT control, floppy disk control, management of the real time clock, and generation of external video signals. A 64 Kbyte buffer is reserved for the system's text editor, the software for which is contained in the main CPU's FORTH dictionary.

The other CPU boards are used as front-end processors for the data streams being analysed, and are capable of both receiving and transmitting on those streams. Each of these boards contain 1 Mbyte of RAM, and their FORTH dictionaries contain the application software for the protocol testing. Any user-programmed software (either additional FORTH words or simulation test scripts) is placed in the RAM of the front-end CPUs.

For large applications, 1 Mbyte may not be sufficient storage for the FORTH dictionary. In this case, a standard memory extension card for the VME bus may be inserted into one of the spare card slots. Software support exists in the basic K1195 FORTH dictionary to allow such a memory extension to be linked to any of the CPU boards. This software renders the

discontinuity of memory addresses between the top of the CPU RAM and the bottom of the memory extension RAM invisible to the programmer. In the case of the ISDN protocol test system, a memory extension card is required to run the software.

Two auxiliary serial RS232 ports are also provided. The first of these is used to connect to a printer, allowing both source file listings and dumps of data traffic to be printed. The data traffic may be printed in any of the formats provided by the monitor function.

The second of these ports can be connected to a remote terminal or network control centre, allowing the K1195 to be operated remotely in a master/slave configuration.

4 FORTH Flavour

The "flavour" of FORTH used on the K1195 is based on FORTH-79, Fig-FORTH, and 68k FORTH with considerable enhancements. These enhancements reflect the specific system design of the K1195 as a self-contained computer, in addition to providing support for data communications. They include disk file system tools, a text editor, display enhancements, and support for timer manipulation, configuration of the test ports, and receipt and transmission of data on these ports.

The structure of a K1195 FORTH dictionary entry is shown in Figure 3. The K1195 uses 32 bits for each address, which requires that each dictionary entry has a code field size and a link field size of four bytes each. If the dictionary entry has been generated by a colon definition, the parameter field will be a multiple of 4 bytes long. To avoid accessing or writing to odd memory locations, the name field is a multiple of two bytes long. The name string, like all strings used on the K1195, is stored with the first byte containing the length of the string, and subsequent bytes containing the string characters. If the entry's name has an even number of characters, (and hence the name string uses an odd number of bytes), an unused byte is placed before the name field to ensure the link field is aligned to an even address.

5 Development Environment

The task of developing software for the K1195 (once a specification and design has been completed) consists of the following steps.

- writing an initial version of source code in FORTH
- compiling this code to add new words to the existent dictionary.
- Debugging the software, and making subsequent modifications to the source code, followed by recompilations.
- Saving a binary image of the dictionary once debugging is complete, which may be copied directly into memory at a later stage.

It is possible to perform all of these steps on the K1195, using the editor, the K1195 filesystem, and the FORTH interpreter. However, due largely to inadequacies in the K1195 filesystem, it

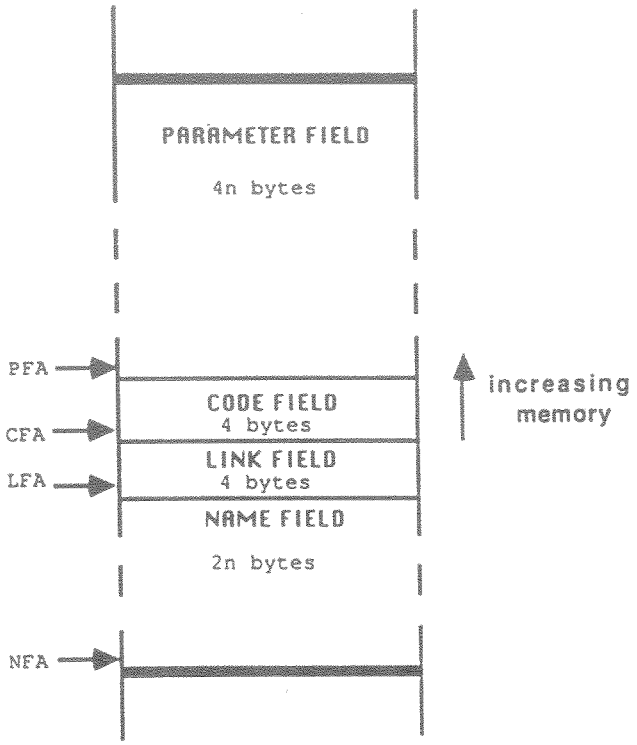


Figure 3. K1195 FORTH dictionary entry

was decided to make use of a UNIX host (initially a VAX 750 running UNIX 4.3 BSD, followed by a microVAX running ULTRIX,) for a large part of the development. The development and subsequent management of the FORTH source code was therefore facilitated by the use of more sophisticated text editors, a hierarchical directory structure, a range of standard UNIX text utilities (such as "grep" for locating occurrences of a string in a file, "sed", a stream editor for multiple changes, etc.), and the RCS revision control system for maintaining different versions (both release and intermediate) of the software.

The K1195 was used to perform compilation and debugging of the software, although it was possible to obtain a FORTH interpreter for the UNIX host. This was because the software made considerable use of the data communication specific enhancements in the standard K1195 FORTH dictionary, which were not available on the UNIX host. Testing on the UNIX host could therefore only verify the correct operation of a small part of the software, leaving any words which manipulated the K1195 hardware untested. In fact, the nature of the development meant that these words were more experimental and hence were likely to contain the most errors!

Files were transferred between the UNIX host and the target K1195s by means of an asynchronous data transfer package called "rfilex". The machines were connected by a direct line between a terminal port on the UNIX host and the remote port on the K1195. In addition to transferring the source files, "rfilex" converted them into the correct format required by the K1195 filesystem, and stored them on 3.5 inch floppy disks. One K1195 was used exclusively for the purpose of up/downloading files, and the disks so produced were compiled on other K1195s.

6 Productivity Aspects

The exponents of FORTH claim that it has compactness, extensibility, an interactive interface, speed and a top down approach. Our experience has found the first three to be useful but the last two to be in some doubt.

The main advantage that using FORTH brought to the project was its interactive interface. This was useful for trying various options and immediately being able to verify the results, and for patching the software without having to do a complete recompile. Of course, anything altered interactively also had to be added to the source code and ultimately recompiled and tested but there was a high probability changes made this way would work first time. Care had to be taken though, to keep track of what changes had been made interactively and to make sure they all did eventually get put in the source code.

It is difficult to compare the compactness of FORTH code with some other language because the software would certainly have been designed differently if say C was used. Also, using C would mean that the overhead of maintaining a dictionary would not be required but a more complex support environment and a means of executing the code would be.

FORTH is certainly extensible. Later sections of this paper will explain features that were added to FORTH to ease the development of this software. There are dangers in this approach though. If the extensions are machine specific, then the software loses its portability and in this case all development and testing had to be performed on the target hardware which was quite expensive. If the extensions are not machine specific, they still add to the final code size and speed and make it harder for another programmer to understand what is going on.

7 Disadvantages of FORTH

The main disadvantage of FORTH on this project (which involved several programmers) was the way that one person's code could inadvertently modify another persons and the lack of any rigid interfaces between different parts of the software. This meant that there was a substantial integration phase that had to be performed, where all the different parts of the software were put together and run. This was usually done just before a deadline was due and it meant that testing time was often spent rewriting software because of a misunderstanding between programmers. If a language such as Modula-2 was used, that forced the interfaces between modules to be explicitly defined, these episodes would probably have been minimised.

FORTH has no range checking, presumably in the interests of speed. This meant that a store operation could be performed using any number on the stack as an address, even if this was illegal or in a part of memory that you had nothing to do with. In the first case, if the address was odd and you were attempting to store a full word, the machine would crash requiring a reboot (this was later modified so that a reboot was not required but you were still returned to the FORTH interpreter from your application). In the second case, the error may not be discovered until the code that was modified was then executed, often only revealing very subtle bugs.

There is an overhead involved every time a FORTH word is called from another FORTH word. The CFA of the called word must be fetched from the PFA of the calling word and transferred to

and the return address saved on the return stack. This overhead proved to be too expensive in a number of cases, resulting in some words being coded directly in assembler. It also meant that FORTH's top down design approach could not always be used because it resulted in too much nesting. In one extreme case (moving a byte from one location to another and post-incrementing the source and destination pointers) can be up to a hundred times slower in FORTH than the single assembly language instruction required.

Also on the subject of design, it is possible to manipulate the return stack and force a word to not return to the word that called it, but to the word that called the calling word or some other word. This is useful sometimes to reduce the amount of nesting of IF statements for example, but when it comes to debugging it makes the debate about GOTOs and "spaghetti code" in BASIC seem trivial. Care must also be taken in jumping out of loops because the return stack is used here as well.

As a final point on the FORTH environment, you can easily be initially impressed by the amount of control the FORTH interpreter gives you over the stack, the return stack and the complete memory and resources of the machine. But after laboriously tracking down bugs that turn out to be simple coding or typing errors, you begin to appreciate the more traditional approaches that take some of this control away from you and perform some of the more tedious tasks such as stack manipulation for you.

8 Generic Design

Telecommunication protocols are usually standardised by international bodies such as the CCITT. However, in many cases, certain design decisions are left to the national telecommunications provider, e.g. Telecom. Differences in protocol specifications are therefore commonplace in different countries.

To be useful for more than one national variant of a protocol, testing software should be designed so that a minimum of source code has to be rewritten to adapt it to a new variant of the protocol. This has led to a design strategy of representing a protocol specification in a descriptive rather than a procedural way i.e. by use of data instead of instructions.

An example of this strategy can be found in the way the structure of data frames is represented. A procedural method for coding a data frame would require a new FORTH word for each national variant of the frame structure. A descriptive method would only require one FORTH word, which would code the frame according to data found in defined tables. Changing the table contents would alter the structure of the coded frame. The table contents may be likened to a blueprint of the frame structure.

It should be pointed out here that a descriptive method involves a specification of how the data in the tables is to be organised, and such a specification limits the differences in the protocol that can be catered for. A simple data organisation can only be used where the protocol differences are minor. In general, as the differences in the variants of the protocol increase, the data representation of the protocol must become more elaborate.

Fortunately, national variants of the ISDN Access and CCS7 protocols are similar enough to allow the descriptive method to be widely used throughout the software without the data organisation becoming prohibitively complex.

FORTH is well suited for both procedural and descriptive methods of protocol specification. Two ways may be used to implement a descriptive method. By using pointers to data tables, and having several versions of these tables in memory, the protocol specification may be changed quickly by altering these pointers to address a different set of tables. A more memory efficient way is to use only one set of tables, obviating the need for pointers, and to overwrite the table contents every time a new protocol specification is required. The disadvantage here is that this can only be done when the software is compiled.

The basic K1195 FORTH dictionary contains the words DOER and MAKE to facilitate the procedural method of protocol specification. When the FORTH interpreter encounters a colon definition, a new word is both declared (it may be used by following words) and defined (its actions are specified). The DOER and MAKE words enable other FORTH words to be declared and defined at different times in the compilation. Such words may then be redefined if required. An example is appropriate here.

A word (CLEAR_CALL)is declared by the FORTH sequence

```
DOER CLEAR_CALL
```

This creates a dictionary entry for the word CLEAR_CALL, which may then be used in subsequent definitions. At this stage, CLEAR_CALL performs no action. At a later stage, this word may be defined to perform some action by using MAKE, for example

```
MAKE CLEAR_CALL
     BREAK_CONNECTION
     FREE_RESOURCES
     RESET_TIMERS
;
```

Now, whenever CLEAR_CALL is invoked, the words BREAK_CONNECTION, FREE_RESOURCES, and RESET_TIMERS will be executed. This is true even for words which invoke CLEAR_CALL which were defined before the MAKE operation.

To change to a different variant of the protocol, in which a call clearing sequence consists of different actions, the following code may be executed

```
MAKE CLEAR_CALL
     SEND_CLEAR_REQUEST
     WAIT_FOR_CLEAR_ACKNOWLEDGE
     FREE_RESOURCES
;
```

Another form of procedural specification is the use of pointers containing the addresses of FORTH words which execute the protocol. If a different protocol behaviour is required, a new word can be defined to execute this, and the pointer reset to contain the address of this new word.

All of the above methods have been used successfully in the protocol testing projects to implement different protocol variants. The choice of one of these methods over another depends on which aspect of the protocol definition is being represented. For example, the use of pointers to data tables was considered the most suitable method of coding data frames, while vectored execution i.e. using arrays of addresses of FORTH words was deemed most suitable for decoding and displaying received frames.

9 C-FORTH Interface

FORTH compares favourably with other high level languages in terms of execution speed. However, in certain parts of protocol testing software, the speed requirements cannot be met by using FORTH alone. This is particularly true when analysis is being performed on a high speed data stream.

To improve the speed performance of a FORTH based program, the number of levels of nesting of FORTH words can be reduced. Ideally, for time critical procedures in the software, this should be reduced to one level by coding the procedure in assembler, and storing the resultant machine code in the parameter field of a FORTH word. The code field of that word should contain the word's PFA. This is the method by which the lowest level FORTH words in the dictionary are defined. The problem here is that while execution time is reduced, the time taken to develop software in assembler is many times that taken using FORTH.

To overcome this problem, a C - Motorola 68000 cross compiler was used on the UNIX host. This approach has the dual advantages of decreasing software development time, while allowing a proportion of the software testing to be done on the UNIX host, taking advantage of debuggers such as "dbx".

The process of running a C program on the K1195 involves three stages:

- Compiling the program into 68000 executable machine code
- Loading the compiled machine code into the K1195 RAM
- generating FORTH dictionary entries to execute the code and access any variables

This process is illustrated in Figure 4.

The distribution of the software between C and FORTH was determined by the speed requirements of individual modules in the software. Time critical modules (generally those which are required to process data while it is being received by the K1195) were developed using C. Other modules were written in FORTH. In addition, the user interface to the software was kept entirely in FORTH to give the end user of the K1195 the full flexibility of the language.

A range of utilities, both in FORTH and C, were developed to provide a usable cross-compile environment. The first of these was a FORTH word designed to load a binary file generated on the UNIX host into the K1195's memory.

```
CLOAD ( filename - - )
```

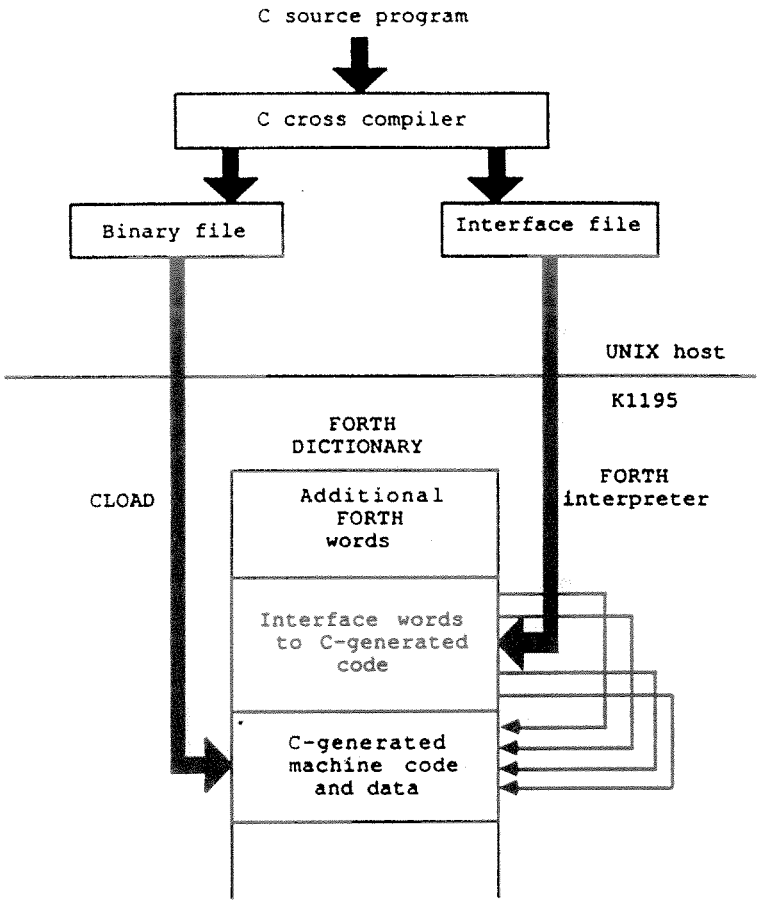


Figure 4. Development process using the C cross compiler.

CLOAD examines the specified file, firstly checking that the magic number in the file header is that of a binary file suitable for execution. If it is, the header is then consulted for the sizes in bytes of the three segments of the program (text, data, and BSS), as well as the address at which the binary is to be loaded. The text and data segments are then copied to this address, and memory is reserved for the BSS segment. Finally, the start address and the three segment sizes are reported to the K1195 screen.

FORTH words are also required to interface to this binary code, i.e. to call procedures and access data. In order to automate this, the C cross compiler was modified so that if the source file being compiled contained the dummy enumerated type

```
enum FORTH_IF;
```

an auxiliary file would be produced. This file contains a line for each variable, pointer, array, function, or structure offset defined in the C program. Each of these lines is of the form

```
number  symbol_type  identifier
```

The identifier is the same as that used in the C program. The number depends on the symbol type. If for example, the symbol type is that of a variable, the number is the address where the variable is stored. If it is a structure offset, the number is the number of bytes to be offset. If it is a function, the number is the address of the entry point to that function.

The following symbol types are currently supported.

SYMBOL TYPE	DESCRIPTION
CL	long variable address (4 bytes)
CW	short variable address (2 bytes)
CC	byte variable address (1 byte)
COL	structure offset to a long variable
COW	structure offset to a long variable
COC	structure offset to a long variable
CFN	entry point to a C-defined function
CB	base address of a structure
CS	structure type
CARY	base address of an array
CPTR	pointer address

To further facilitate the use of the cross compiler, FORTH words were defined for each of the supported symbol types. These words are compiling words, and allow the auxiliary file to be used directly as a FORTH source file to generate the required interface words. For example, executing the line

```
0x68000 CL variable_name
```

will cause a dictionary entry of name "variable_name" to be created, the run-time action of which will be to return 68000 (hex) on the stack, in much the same way as a standard variable returns the address where the value is stored. Executing the line

```
0x68100 CFN function_name
```

will cause a dictionary entry of name "function_name" to be created, the run-time action of which will be to begin execution of the machine code starting at address 68100 (hex). Executing the line

```
20 COL offset_name
```

will cause a dictionary entry of name "offset_name" to be created, the run-time action of which will be to add 20 to the value on the top of the stack, thereby offsetting a base address of a structure to generate the address of one of its elements.

This process automatically generates an interface word for each function, variable, structure base, structure generator, structure offset, array, and pointer declared in the C program. If FORTH interface words are not required for all of these, the appropriate lines may be deleted from the auxiliary file prior to downloading to the K1195.

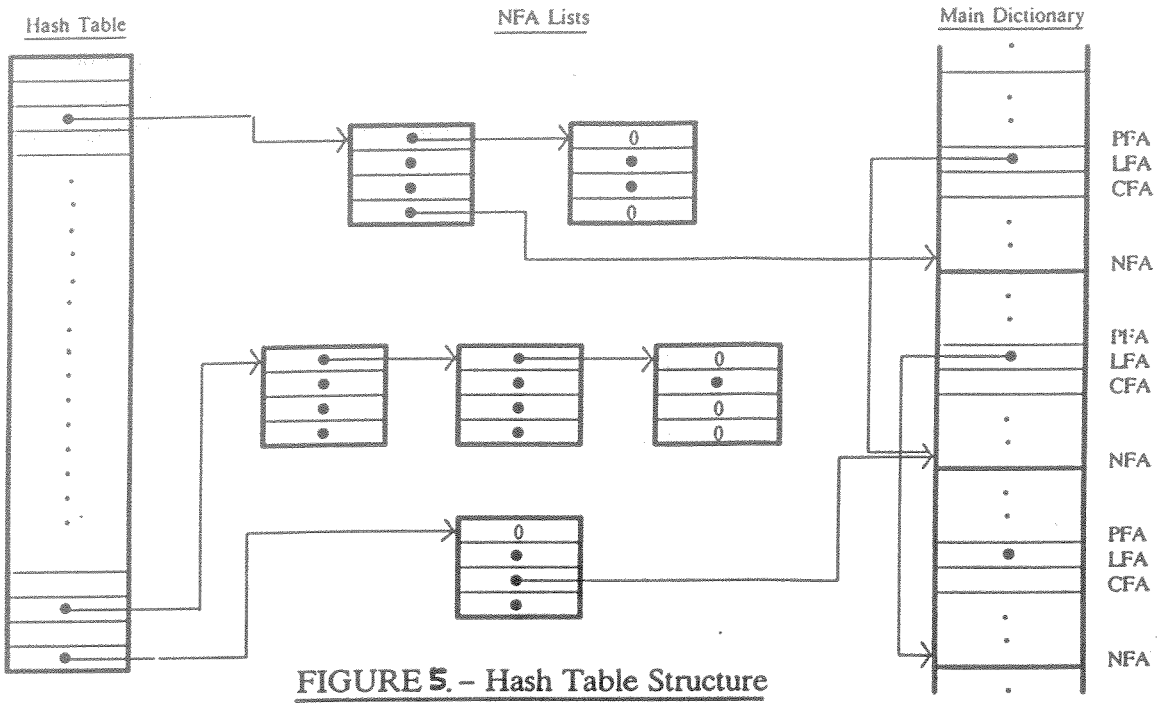


FIGURE 5. – Hash Table Structure

10 FORTH Software Development Utilities

During the course of the protocol testing projects, a range of utilities have been developed in FORTH which have become part of the standard K1195 FORTH dictionary. These utilities have enabled the software development to proceed in a faster, more efficient, more structured, and less error-prone manner. As such, they serve as excellent examples of how the extensibility of FORTH enables the programmer to improve the language and overcome many of the deficiencies of standard FORTH.

10.1 Fast Compiler

When the FORTH interpreter compiles a colon-defined word, it must search the dictionary to find each of the component words of the definition, so that their Code Field Addresses (CFAs) may be placed in the Parameter Field of the new word. This process originally employed a linear search algorithm, which became slower as the compilation proceeded and more words were added to the dictionary.

By using a hash table to improve the search algorithm, the compilation time for a large application (the ISDN protocol test software) was reduced by approximately two thirds. When a word is encountered, it is only compared to those dictionary entries whose names are hashed to the same value. The Name Field Addresses (NFAs) of these words are chained together as shown in Figure 5. When a new word is added to the dictionary, the hash function is evaluated using its name, and its NFA is then added to the head of the appropriate chain. The number of NFAs stored in each link of the chain (three) was chosen to minimise the amount of storage required by the hash table and linked lists for a given number of dictionary entries.

10.2 C-like Structures

In order to allow more readable FORTH code to be produced, a series of words were defined to implement C-like data structures. In addition, these words allowed multiple instances of a set of data without the overhead of a new set of FORTH variables for each instance. These words are as follows.

```
STRUCTURE       ( - - 0 )
END_STRUCTURE   ( n - - )

CHAR            ( n - - n + 1 )
SHORT           ( n - - n + 2 )
LONG            ( n - - n + 4 )
CHAR_ARRAY      ( n \ m - - n + m )
SHORT_ARRAY     ( n \ m - - n + 2*m )
LONG_ARRAY      ( n \ m - - n + 4*m )
```

The usage of these words is best explained by example. Consider the following

```
STRUCTURE  DATE             ( define the word DATE to be a structure )
                            ( generator with the following contents )
        LONG DAY            ( 4 bytes )
        LONG MONTH          ( 4 bytes )
        LONG YEAR           ( 4 bytes )
        8 CHAR_ARRAY DAY_NAME       ( 7 bytes )
        4 CHAR_ARRAY MONTH_NAME     ( 4 bytes )
END_STRUCTURE               ( end the structure definition )
```

The code between the words STRUCTURE and END_STRUCTURE defines the size of the structure. This is done by updating the top element of the stack as each structure element is encountered. When the structure definition is complete, the size is assigned to the word DATE. The following line

```
DATE TODAYS_DATE            ( generate an instance of the structure )
```

then generates a dictionary entry with name TODAYS_DATE, and reserves enough memory to fit an instance of the structure. When TODAYS_DATE is executed, the base address of this memory is returned. The element operators, DAY, MONTH, etc., then add the appropriate offset to this base address, thereby returning the address of the required element. The following lines of code

```
1988 TODAYS_DATE YEAR  !    ( set the year to 1988 )
     TODAYS_DATE MONTH @    ( return the month )
```

illustrate the use of these operators to access the required structure elements.

10.3 Modules

One of the greatest criticisms levelled at FORTH is its lack of local variables and procedures i.e. every variable and procedure is global and may be accessed by any subsequent FORTH word. This property of FORTH prevents good software modularisation, where only a well defined interface to each module is visible to other modules.

To overcome this deficiency, the following words were defined.

```
MODULE
END_MODULE
EXPORT
```

The use of these words is illustrated below.

```
MODULE module_name

( definitions of FORTH words internal to this module )

        EXPORT word_1
        EXPORT word_2
        EXPORT word_3
END_MODULE
```

The word MODULE marks the current top of the dictionary for later use by EXPORT. EXPORT is used to specify which of the words defined in this module are to be visible to other subsequent FORTH words. This is done by linking the following word to the previous word specified by EXPORT, or, if the word is the first to be exported in the current module, by linking it to the last word defined before the module was begun. END_MODULE causes the last word specified by EXPORT to be marked as the latest defined word in the dictionary, so that the next "normal" word defined will be linked to it. The implementation of these words requires all the EXPORT operations to be done just prior to the END_MODULE word.

10.4 Fast buffer routines

Timing analysis of protocol testing software shows that a considerable amount of time is spent manipulating data buffers. One example of this is when a frame is being decoded, and the buffer it is stored in is scanned, byte by byte, while each field of the frame is extracted and displayed. Another is when a frame is being coded for transmission. Here, each byte is coded and placed in the next available location in the transmission buffer.

The following words were defined to facilitate buffer operations. They were coded in assembler to optimise their speed. Their use has lead to speed improvements of 20-30% in typical frame decoding/coding operations.

```
CP_SET ( Buffer Start Address \ Count --- )
```

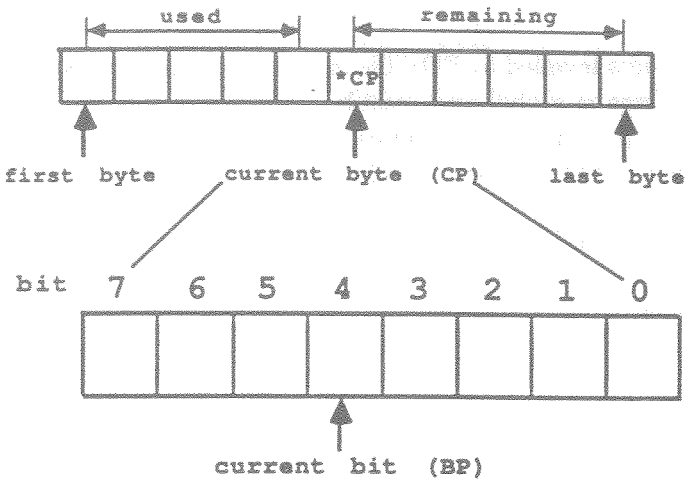


Figure 6. Use of pointers in fast buffer routines.

Initialize four internal system variables which will enable the specified buffer to be managed by the system. The current pointer will be set to the start of the buffer and the Bit Pointer will be set to bit 0.

CP_START (---)
Will move the current position to the start of the buffer.

CP_END (---)
Will move the current position to the end of the buffer.

?CP_REMAIN (--- COUNT)
Returns the remaining length of the current buffer.
(See Figure 6.)

?CP_USED (--- COUNT)
Returns the used length of the current buffer.
(See Figure 6.)

CP (--- POINTER)
Returns address of the current position within the managed buffer.

NEWCP (pointer --- flag)
Sets current position to the supplied address if it is within the buffer. A -1 is returned if the new position was within the buffer, or a 0 is returned if the position was outside the buffer and the current position wasn't changed.

CP++ (--- flag)
Tries to move the current position within the buffer one byte towards the end of the buffer. A -1 is returned on success.

CP-- (--- flag)

```
     Tries to move the current position within the buffer one
     byte towards the start of the buffer. A -1 is returned on success.

+CP ( OFFSET --- flag )
     Tries to move the current position by OFFSET bytes. A positive
     OFFSET will move the current position towards the end of the buffer.
     If the resulting position is within the managed buffer a -1
     is returned on stack, if not, the position is unchanged
     and a 0 is returned.

@CP ( --- BYTE )
     Returns the byte from the current position within the managed buffer.

!CP ( BYTE --- )
     Stores the byte on the stack in the current position of the managed buffer.

@CP++ ( --- BYTE \ flag )
!CP++ ( BYTE --- flag )
@CP-- ( --- BYTE \ flag )
!CP-- ( BYTE --- flag )
     These words combine the byte fetch or store operations with a
     post-decrement or post-increment on the current position pointer.
```

10.5 Fast Bit Stream Routines

The byte operated on is that at the current position within the managed buffer. The Bit Pointer (BP) is set to Bit 0 whenever the current pointer is moved.

```
BP ( --- bit number )
     Returns the bit number currently pointed to. ( see Figure 6. )

BP++ ( --- )
     Tries to move the Bit Pointer within the byte one bit towards Bit 7.
     The pointer will not be moved past Bit 7.

BP-- ( --- )
     Tries to move the Bit Pointer within the byte one bit towards Bit 0.
     The pointer will not be moved past Bit 0.

+BP ( OFFSET --- )
     Tries to move the Bit Pointer by OFFSET bits. A positive OFFSET will
     move the Bit Pointer towards Bit 7 and a negative OFFSET will move
     towards Bit 0.  The pointer will not be moved past either Bit 7 or
     Bit 0.

@BP++    ( number bits --- bits )
     Will fetch  the number of bits specified from the current
     Bit Pointer position.
```

```
    The Bit Pointer is moved to point at the bit following the last
    bit fetched but will not be moved past Bit 7.

!BP++    ( bits \ number bits --- )
    Will store the number of bits from the low order bits of the supplied
    value at the current Bit Pointer position.  The Bit pointer will be
    moved to point at the bit following the last bit stored, but will
    not be moved past Bit 7.
```

10.6 String table generators

Standard FORTH allows tables to be conveniently generated by use of the , (comma) word. This word stores the top stack value on the top of the dictionary, and advances the dictionary pointer. In the following example, a table is generated containing the values 1, 2, 3, and 4. The base address of this table is returned by the word TABLE_START.

```
CREATE TABLE_START 1 , 2 , 3 , 4 ,
```

However, creating a table of string addresses is more complicated, as the strings themselves must be stored somewhere. Ideally, the strings should be stored starting directly above the table, but the size of the table is not known until the final table entry is declared. To facilitate this procedure, the following words have been defined.

```
INIT_STR_TAB
END_STR_TAB
,"
```

INIT_STR_TAB is used to reset a pointer to the start of an unused portion of memory. Each string which is to have its address stored in the table is then declared using the ," word, as in ," string" . Note that the string is delimited by the next occurrence of the " character. As each string is encountered, it is temporarily stored in the unused portion of memory, along with the address of the table entry which will eventually contain the string's address. The dictionary pointer is advanced by one table location, but nothing is stored in the table yet. When END_STR_TAB is encountered, all the strings are copied directly above the top of the table, and their starting addresses are stored in the table. The following example illustrates the use of these words.

```
CREATE TABLE_START
INIT_STR_TAB
    ," string1"
    ," string2"
    ," string3"
END_STR_TAB
```

The dictionary pointer is advanced to just past the last byte in the last string. In this manner, the strings themselves become embedded in the dictionary.

10.7 Debugging utilities

The following words form part of the standard K1195 dictionary, and have been used extensively to assist in the debugging of the protocol testing software.

```
FIND ( - - )
        Usage: FIND word
        Reports all dictionary entries which use the specified word.

DECOMP ( - - )
        Usage: DECOMP word
        Reports all the component words used in the definition of the
        specified word.

ALTER ( - - )
FROM  ( - - )
TO    ( - - )
        Usage: ALTER word1 FROM word2 TO word3
        Causes the first occurrence of word2 in the definition of word1
        to be overwritten by word3.

UNALTER ( - - )
        Usage: UNALTER
        Causes the last ALTER operation to be reversed.

NULLIFY ( - - )
        Usage: NULLIFY word
        Alters the definition of the specified word so that it does
        nothing.

CHANGE ( - - )
INTO   ( - - )
        Usage: CHANGE word1 INTO word2
        Alters the definition of word1 so that it executes word2 only.

SEARCH ( string_address - - )
        Usage: " string" SEARCH
        Reports all words in the dictionary which contain the given
        string as part of their names.
```

11 Conclusions

FORTH appears to have limitations when it is used in a large, multi-programmer environment. There is insufficient protection of one persons code from modification by someone else's and there is no formal, enforceable method of defining the interfaces between modules of code.

However, FORTH does have definite advantages in the prototyping environment. The interpretive interface and the ability to modify the code currently in memory make quick "hacks" and

trial and error solutions quite feasible. Also, FORTH's extensibility allows new features to be added to the language and these can overcome some of the above problems, but they make the code less portable. Even when C was used for its speed, the FORTH interface was retained for these advantages.

12 References

1. K1195 User Manual, Siemens AG
2. Leo Brodie - Starting FORTH
   Prentice-Hall Inc., 1987
2. Anita Anderson, Martin Tracy - Mastering FORTH
   Brady Communications Company, 1984.
4. S.Leask - ISDN Access Signalling Emulator
   IREECON, Sydney, September 1987.
5. B.Dingle, W.Bartelt - ISDN Access Testing
   IREECON, Sydney, September 1987.

13 Acknowledgement

The permission of the Chief General Manager, Telecom Australia to publish this paper is hereby acknowledged.

— End —
March 31, 1988