FORTH AND PARALLEL PROCESSING.

CONTROL FLOW, REDUCTION AND DATA FLOW COMPUTERS.

A computing machine is normally conceived as a CONTROL FLOW machine. Such a computer is driven by the program; the program determines the timing, the movement of data, scheduling of output, etc. There are other possible styles of computer and two concepts are commonly used: REDUCTION machines and DATA FLOW machines (Treleaven et al, 1982).

Reduction machines are often also called 'demand driven', because the operation of the computer is determined by requests for output, rather than by the structure of the program. When a reduction machine is operating, all programs are in an idle state until an 'answer' is requested. Then, the software operation which controls output looks to see what input it requires, and that input is demanded from the previous operation. Each operation demands input data until, eventually, all requisite data are tracked to their sources, and the program begins to execute. The data flow through the machine, all the while being modified by the operations along the way. Eventually the 'answer' is supplied. The demand driven concept sounds as if it would be inefficient, but it can be made efficient, and it is very similar to the operation of a modern Just-In-Time factory.

Combining elements of control flow and demand driven, the third concept - data flow computing - is the most interesting of the three. With data flow computing the program comprises a number of independent processes which are able to pass parameters to each other. Each process is idle until all its input requirements are satisfied, whereupon it executes and produces output. As each process executes the data ripple through the program until an 'answer' is generated.

Data flow computing concepts have been very important in the development of parallel computing machines and software, although no really successful pure data flow computer has ever been built. (Gottlieb et al, 1983, p.34) The Manchester University data flow computer is probable the most successful experimental machine in existence. (Watson and Gurd, 1982) (Gurd et al, 1985)

To examine data flow more closely, consider the example shown in Figure 1., a program to calculate

$$z = (x + y)(x - y)$$

(Dennis, 1980). There is both parallelism and sequentialism in this calculation. (x+y) may be calculated before (x-y) or after it; the order of

execution of these processes does not matter. They may even be calculated on different computers. However both the (x+y) and the (x-y) calculations must be complete before z can be calculated. Although this is a trivial example, it demonstrates that it would be possible to make the computation on 1, 2 or 3 independent computers, provided the parameters were passed from one to the other in an orderly fashion.

It should be explained that each 'process' in data flow can be as simple as one instruction, or even a NO-OPERATION, or can be as complex as a complete software package. True parallel computing consists of one job or problem, being partitioned into a number of processes which are executed in some predetermined parallel/ sequential order. If the job is partitioned into a large number of small processes the parallelism is called FINE-GRAINED, and if it is partitioned into a small number of complex processes it is described as COURSE-GRAINED. It used to be thought that it was desirable to have as fine a grain as possible (Dennis, 1980, p.48), but experience with simulations has shown that the software overhead expands alarmingly with the fineness of grain, and this has been just one of the reasons for the lack of success of data flow computers. It is now accepted that course grained parallelism works well (Kruatrachue & Lewis, 1988).

If processes in a program are formed into groups which are data independent, then there are several jobs running concurrently, and the computer is then multitasking. Multitasking, or multiuser computing is merely a subset of the wider picture of parallel computing. A parallel program may be executing on several PROCESSORS, and the number of processes may be greater (not less) than the number of processors. So parallel computing implies concurrency - several processes running or idling simultaneously on each processor.

True data flow computing has strict semantics which have been developed by a number of researchers (Ackerman, 1982) (Arvind & Gostelow, 1982). The semantics determine the firing rules (when a process executes it is said to FIRE). The firing rules are necessary for an ordered progression of the program. The semantics of data flow are unimportant here. In simplified terms, the useful firing rules are:

1. A process may only fire if all input parameters are available.
2. A process may not fire if any of its output parameters have not been absorbed by succeeding processes.

PARALLELISM WITH FORTH

In order to make a concurrent/parallel computer in the data flow style, work for FORTH, some other components are necessary. To prevent the program HANGING UP it is

desirable to have no endless loops in any process, and to time slice the execution of the processes finely. Task switching in the interpreter is therefore inadequate, because the processor may spend too long with one task or one user. In addition, parameter passing through the stack does not work, because of its sequential nature, so some other data structure needs to be used. For a simple multitasking FORTH all that is required is a straightforward task switching program which has executive power over any of the Forth programs operating concurrently. This task switcher would store parameters in a reserved area of memory, accessed only by itself.

A task switcher for a concurrent Forth for the Motorola 68000 was written in assembly language. For each Forth process only two parameters are passed: the return stack pointer and the user stack pointer. Naturally, each individual Forth process has its own pair of stacks. All other user variables have been specified as global and can not be changed by any one Forth process. For example, it is not possible to have one process operating in decimal, and another in hexadecimal, because BASE is a global variable in this implementation.

The stack pointers are kept in a queue and the executive operates as follows:

1. Stop execution of FORTH.
2. store return stack pointer and user stack pointer on end of the queue, using queue tail pointer
3. Check if at the end of queue memory space.
4. If so, index queue tail pointer to top of memory space
5. Use queue head pointer to obtain new return stack pointer and user stack pointer.
6. Check if queue head pointer at end of queue memory space.
7. If so, index to top of queue memory space.
8. Continue with FORTH.

The task switcher is inserted into the NEXT routine which is part of SEMIS (;S), so that the task is switched whenever a high level Forth word terminates, but not on termination of a primitive. It is necessary to rewrite, in high level Forth, any primitives which may contain endless loops. Such words might be EMIT and KEY. This is to prevent hang-up.

A concurrent Forth like this is now successfully executing process control programs at a number of installations in Queensland. These programs are dedicated, and only permit connection of one terminal, so they are multitasking, single user. The programs are also programmed into Read Only Memory and do not use disk drives.

For complex parameter passing and concurrency between one or more Forths and other processes written in other

languages a more formal method is required. The process itself may be represented by a data structure in Random Access Memory. This structure contains data, pointers, flag bits, and data counters. One such structure (called an I-structure by Arvind) (Gajski & Pier, 1985, p.21) (it is also similar to the 'activity templates' of Dennis) (Dennis, 1980) for the MC 68000 is as follows:

```
  32 bits   SWITCH FIELD - pointer to executing code
  16 bits   X FIELD - number of input variables
  16 bits   Y FIELD - count of input variable arrivals
first datum
  16 bits   datum empty/full bit and pointer to switch
            field of this structure.
  32 bits   DATUM itself.
second datum
  16 bits   } as above
  32 bits   } as above
any other data
output destinations
  32 bits   FIRST destination
  32 bits   SECOND destination
other destinations
```

For this arrangement to work, each datum is fitted with a destination address to form a 64 bit PACKET. The packets are injected into a queue in the same way as was described previously. When a packet is popped from the queue the executive reads its destination address and looks at the empty/full bit. If the slot is empty the datum is inserted and the Y field is incremented. The executive then checks to see if the Y field matches the X field, and, if it does it executes the program pointed to by the switch field. The switch field is found from the pointer after the E/F bit. If the destination is full already, the packet is returned to the end of the queue.

Once a process has executed, the output data are transmitted to the end of the queue and the executive is invoked again. In the case of multitasking Forths, only two parameters, the return and the users stack pointers, need ever pass through the queue. Any other parameters to be passed between the Forths may either be passed in the same way, (safe method), or via a known memory address (hazardous method). If other variables such as BASE, CONTEXT, CURRENT are varied for different tasks, they must also be included amongst the parameters to be passed. Only one copy of the Forth kernel is used.

A problem arises, with this arrangement, if data are being input to the program faster than they are being absorbed, in which case the queue will expand and eventually overflow. The reason for this is that only the first of the two firing rules mentioned earlier is invoked. The second firing rule prevents conjestion of packets between processes.

To implement the second firing rule, output data can be stored in the I-structure instead of in a queue, and any process is inhibited from firing if there is processed output awaiting transmission. The executive then merely keeps a list of I structure output registers where it can find the packets. Each full output register must be visited in turn, without favour, to see if the packet can be transmitted to the destination structure.

Using such an arrangement it is possible to have one or more Forths with, possibly, varying degrees of autonomy, interacting in an orderly way with other processes, running on the same or other processors. One way of organising such a scheme is shown in Figure 2.

INTERACTION WITH TRANSPUTERS.

The Transputer is a British made processor specifically designed to execute parallel programs very efficiently. It is a 32 bit Reduced-Instruction-Set processor and the integer version - the T414 - runs at 10 MIPs (million instructions per second). The floating point version - the T800 - executes floating point instructions faster than integer instructions. The Transputer is normally programmed in OCCAM (May & Shepherd, 1985), a special parallel language, but compilers for C, Fortran and Pascal are obtainable.

Transputers communicate directly with one another through LINKS which are bi-directional 10 MHz serial connections, which transmit 8 bits at a time. By constructing a printed circuit card with link chips on it, it is an easy matter to communicate between a Forth machine and a Transputer programmed in OCCAM.

In a series of experiments, Forths running on an MC 6809 based computer have been accepting data from computations executing on Transputers, and driving hardware interface circuits. This arrangement is part of a simulation computer, which solves differential equations at high speed and provides output data to analogue recording instruments. The Forth computer has also been used for commissioning and troubleshooting the Transputers, as the Forth interpreter allows a simple method of writing test programs. In addition the Forth has been used for 'capturing' data passed between two Transputers, by interposing the Forth machine between the Transputers. Any byte arriving over a link is passed on, but is also copied out through an RS 232 port to a personal computer which stores the data on a disk file. This was necessary to examine the information being passed through the links. Figure 3 shows the connection diagram of this arrangement.

A PARALLEL FORTH COMPUTER.

By implementing a concurrent version of Forth in the manner described earlier on a number of machines, and by connecting those machines together with the link interface cards, it is possible to construct a highly parallel Forth machine. This has not been done for good reason. The task switcher is executed in software and consumes a considerable amount of processing time, although no benchmark timings have been carried out. The processor chips for which the Forth has been written are very slow in comparison with the Transputer. The main reason for turning to parallel computing is to obtain more speed. The Transputer is exceptionally fast, and the task switching is cast in silicon, so it does not consume any software overhead.

Nevertheless, Forth is a useful assistant, as it provides a simple means of accessing various parts of a complex parallel computer. The computer engineer may look into the workings of a parallel network from a terminal, or may use Forth to supply input or output data in various forms.

REFERENCES

Ackerman, W.B. (1982). Data Flow Languages, IEEE Computer Vol 15, Number 2, pp. 15-24, February 1982.

Arvind, Gostelow, K.P. (1982). The U-Interpreter, IEEE Computer, Vol 15, Number 2, pp. 42-49, Februaryy 1982.

Dennis, J.B., (1980). Data Flow Supercomputers. IEEE Computer, Vol 13, Number 11, November 1980.

Gajski, D.D., Pier, J.K. (1985). Essential Issues in Multiprocessor Systems. IEEE Computer, Vol 18, Number 6, June 1985.

Gottlieb, A., Grishnan, R., Kruskal, C., McAuliffe, K., Rudolph, L., Snir, M. (1983). The NYU Ultracomputer - Designing a MIMD Shared Memory Parallel Computer. IEEE Trans. Computers, Vol C-32, Number 2, February 1983.

Gurd, J.R., Kirkham, C.C., Watson, I. (1985) The Manchester Prototype Data Flow Computer. Communications of the A.C.M. Vol 28, Number 1, January 1985.

Kruatrachue, B. and Lewis, T. (1988) Grain-Size Determination for Parallel Processing. IEEE Software, Vol 5, Number 1, January 1988.

Lerner, E.J. (1984). Data-Flow Architecture, IEEE Spectrum, pp. 57-62, April 1984.

May, D. & Shepherd, R. (1985) Occam and the Transputer. Concurrent Languages in Distributed Systems. Reijns, G.L. & Dalgleish, E.L. (Editors). Elsevier Science Publishers B.V., North-Holland.

Trelevan, P.C., Brownbridge, D.R., Hopkins, R.P., Data-Driven and Demand-Driven Computer Architecture, ACM Computing Surveys, Vol 14, Number 1 January 1982.

Watson, I. and Gurd, J., (1982). A Practical Data Flow Computer. IEEE Computer, Vol 15, Number 2, February 1982.

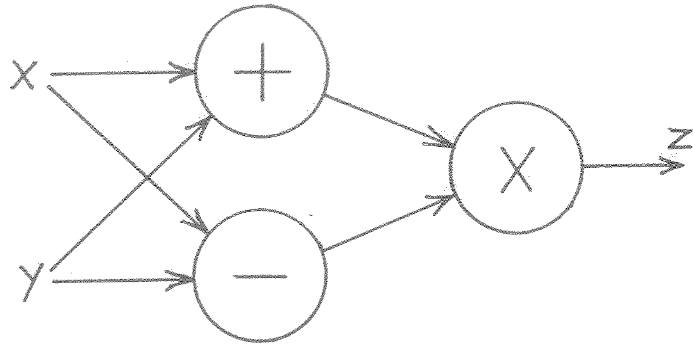

FIGURE 1. Data Flow Graph for
$Z = (x + y)(x - y)$

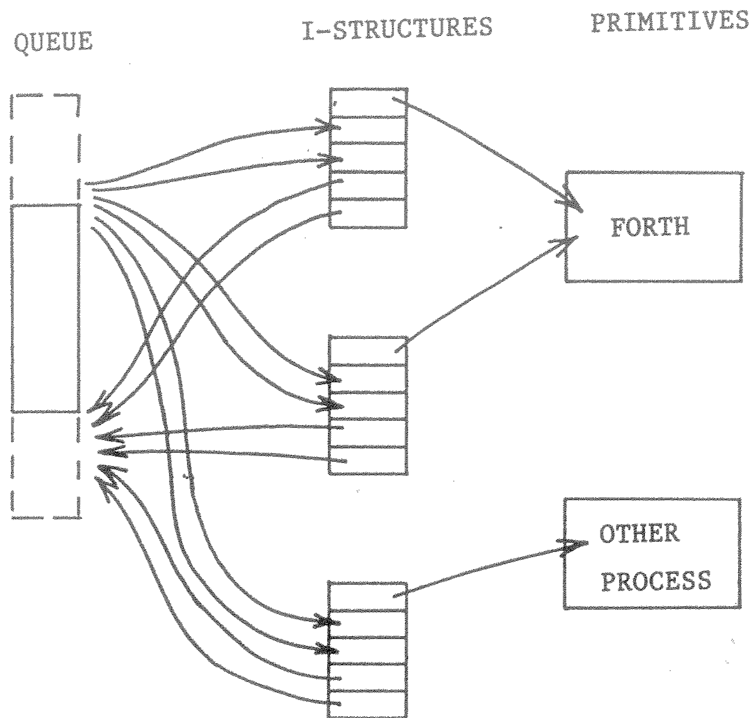

FIGURE 2. Method of Multitasking Forth

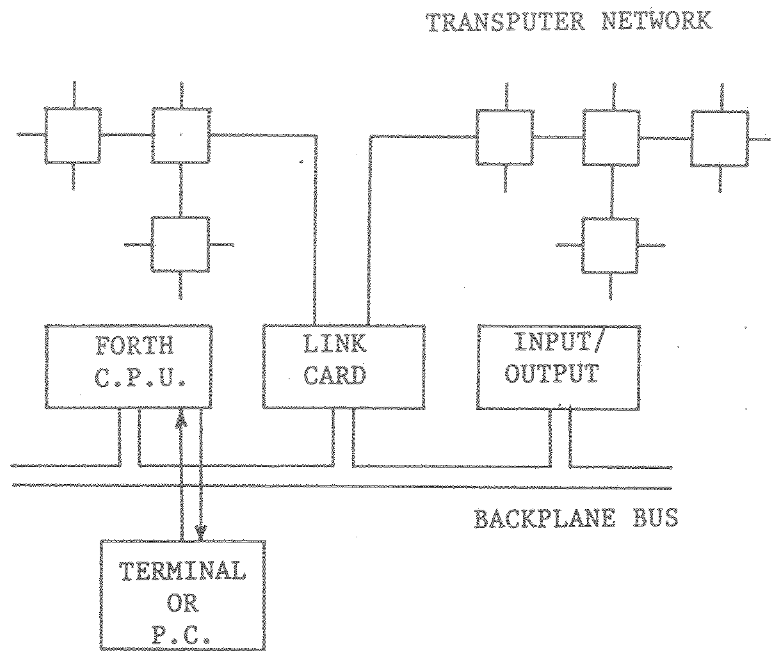

FIGURE 3. Forth in a Parallel Computer

ASYST Applications

Michael G. Smart
Science and Computing Applications Pty. Ltd.

Abstract

ASYST is a FORTH-based language especially suited to engineering and scientific applications. In this paper I will discuss the primary productivity-related features of ASYST (many of them arising from its relationship to FORTH architecture). To illustrate the various features, I will refer to a recently completed Test Management and Data Acquisition System implemented in ASYST for the Traffic Authority of New South Wales.

Introduction

Relationship of ASYST to FORTH

ASYST is a commercially available software development environment for PC-compatible machines which is tailored to scientific and engineering applications. Like FORTH, it uses threaded code, offers both interactive and compiled modes of operation, relies heavily on LIFO stack structures, and allows the programmer to be very productive. In this paper I will outline the particular features of ASYST (most of them common to FORTH) which account for this high productivity.

The similarity to FORTH, which extends to the colon definition syntax for coding subroutines, to stack manipulation commands, to the postfix notation, and many other features, arises because ASYST is itself implemented in FORTH. It may be helpful to think of ASYST as a library of FORTH extensions and enhancements. In particular, these extensions include:

1) Data acquisition hardware drivers for a wide variety of 3rd-party add-on PC boards.
2) A library of data acquisition utilities.
3) A powerful and flexible set of graphics words.
4) An extensive library of mathematical analysis routines.
5) A concise set of array manipulation commands.
6) Floating-point arithmetic.
7) Drivers for many other I/O devices, especially GPIB.
8) Tools for building menus and turnkey systems.
9) Facility to execute assembler programs.

Nature of the Application

The Crash Engineering Unit of the New South Wales Traffic Authority tests seatbelts, motorcycle helmets, child restraints, and other automobile safety equipment. The test apparati simulate high-speed collisions and other vigorous dynamic conditions. The data acquisition problem consists of collecting data from accelerometers and other transducers while the collision is in progress. The principal difficulties arise from acquiring data at a sufficiently high rate and beginning the high-speed acquisition when the collision begins.

Apart from the data acquisition problem, there were two other major requirements of the software: stage-managing a complex sequence of events involving safety procedures, lights, cameras, and a bevy of auxiliary equipment, and making the entire system flexible enough that the operator could conveniently modify test procedures. Let us call these the stage-management problem and the flexibility problem.

The test management and data acquisition system delivered to the Traffic Authority was built around the 80286-based computing platform and ASYST. The data acquisition needs were met with a Data Translation 2821-G high-speed A/D conversion board, a 2 Megabyte extended memory board, and a specialized piece of assembler code which was called from ASYST. The stage-management needs were met by a system (Opto-22) of intelligent, optically isolated digital inputs and outputs interfaced to the computer over a serial line. The flexibility needs were met by a database management front end, implemented in dBASE III+, and by custom-built software-controllable anti-aliasing filters.

Productivity benefits in using ASYST

Planning and Coding

In undertaking a large development project it is necessary to begin coding from the top down. Often this top-down approach begins with a description, in pseudo code, of what the program must do. As the pseudo code is refined and takes on a more detailed character, it more closely resembles the final code.

ASYST (like FORTH) is particularly well suited to this approach because its handling of subroutine calls makes high-level software design as simple as outlining the program's functions. A WORD (the name for ASYST subroutines) is specified in a colon definition. The syntax is typified by the code for the Traffic Authority main program:

```
: MAIN.PROGRAM
    INITIALIZE.HARDWARE
    READ.SETUPS
    ACQUIRE
    ANALYZE
;
```

To invoke a WORD, either interactively or within another colon definition, it is only necessary to give its name. If arguments to the WORDS are desired, the values may be left on the number stack just prior to invoking the WORD. However, if arguments are not required, the subroutine call can be accomplished with great convenience as well as brevity.

A large software system is created in top-down fashion by detailing a fundamental WORD. In the colon definition for this fundamental WORD, a set of new, as-yet undefined, WORDS are invoked. In the colon definitions for these WORDS, new WORDS are invoked at the next lower level of the hierarchy. This process of refinement continues until the WORDS invoked by the colon definitions at one level are all either valid ASYST WORDS or composites of them. Once this point has been reached, the system can be compiled because all WORDS are defined. Not only is the code complete (up to pre-compilation stage), you have created a software structure diagram in outline form.

Having written the above colon definition for MAIN.PROGRAM, one would proceed with top-down coding by writing definitions for the WORDS named:

```
: INITIALIZE.HARDWARE
    INIT.DT2821-G
    INIT.OPTO-22
    INIT.FILTERS
```

```
: READ.SETUPS
   READ.MECHANICAL.SETUPS
   READ.PHOTOGRAPHIC.SETUPS
   READ.ELECTRICAL.SETUPS
;

: ACQUIRE
   CALIBRATE.FILTERS
   CHECK.SWITCHES
   BRING.SLED.TO.FIRING.POSITION
   FIRE.AND.COLLECT.DATA
;

: ANALYZE
   DRAW.TIME.SERIES.ON.CRT
   PLOT.TIME.SERIES
   COMPILE.STATISTICS
;
```

In the actual code, these four colon definitions would have to precede the definition of MAIN.PROGRAM. Any type declarations such as:

```
INTEGER DIM[ 2000 ] ARRAY TIME.SERIES
```

would have to go first so that each WORD is defined before it is first used in a colon definition.

The advantages of this approach to system design are many. Perhaps the most salient ones are that the process is fast because it is not burdened by awkward syntax in the subroutine calls, you create an outline as you go, and it is not necessary to translate your pseudo code into valid statements. The pseudo code becomes the actual ASYST code. Generating executable code is a natural extension of planning the software.

<u>Debugging code</u>

Although it is advisable to begin a software project with a top-down approach, good software is usually the result of several iterations of top-down, then bottom-up, then top-down again, etc. The reason for the bottom-up stages is that you must test program components before you can test the whole. Here we come to the second great strength of the ASYST (and FORTH) programming environments.

In stark contrast to languages such as FORTRAN, Pascal, and C, it is possible in ASYST (and in FORTH) to execute subroutines individually in interactive mode. The convenience this offers the programmer can hardly be emphasized strongly enough. To exercise a subroutine individually in FORTRAN, it might be necessary to write an entire new main program simply for diagnostic purposes.

Even if a symbolic debugger were available, it would still be necessary to execute the entire program to examine just a part of it.

The marvellous simplicity of executing a WORD interactively by typing its name makes a new style of debugging programs possible. Components can be tested one at a time. If one WORD gives an unexpected result, then it can be dissembled into its component WORDS. Each of these can be tested then dissembled until the culprit is ultimately found.

Surprisingly, another serious hindrance to convenient debugging is the excessive use of variables to communicate data between the parts of a program. In the usual case (typified by Pascal, C, and FORTRAN) a calling routine passes data to a subroutine either implicitly, through common blocks or global variables, or explicitly, through arguments to the subroutine call. Such transactions require variables, which must previously have been declared and subsequently initialized, and which may be modified many times by the program.

The debugging problem is that to test a subroutine you must have a clear idea what data goes into it. If the data is supplied by variables, you must be able to examine them in run-time. Without a symbolic debugger one must insert print statements into the code and suffer the attendant inconveniences. Even with a debugger it is not always practical to monitor the flow of data through a variable--especially if the variable is an array.

ASYST employs an alternative method of passing data to subroutines, which solves these debugging difficulties. The values to be passed are placed atop the stack immediately prior to execution of the WORD (the subroutine call). Thus ASYST achieves the same effect as Pascal subroutine calls with VAL (as opposed to VAR) arguments, with two important differences: 1) the programmer can see precisely what values are passed by examining the stack prior to the WORD'S execution, and 2) no variables need be declared.

The combination of stack-oriented operations and interactive WORD execution make for highly efficient debugging. The Traffic Authority system, for example, comprised more than 140 Kbytes of ASYST source code which had to interface two computers to each other, drive nine peripheral controller boards, support three externally supplied software units, and control three hardware units requiring custom software. Debugging this system to the acceptance test stage took three man-weeks.

Interfacing devices

So far, I have discussed features shared by both FORTH and ASYST which account for productive programming. We come now to some device-specific features of the ASYST environment. These features are not shared by FORTH because FORTH, like C, is a compact, portable language incorporating a high degree of machine-independence.

In the Traffic Authority system, the computer communicates with its human masters and with laboratory equipment through a number of I/O modules. These include the following PC add-on boards and associated peripherals:

```
GPIB            digital multimeter
RS422           Opto-22 system
RS232           plotter

Data acquisition/control cards:
  DT2821-G      filters, high-speed input
  DT2801        digital control, low-speed input
```

The ASYST package has inbuilt hardware drivers for these boards (and many others). The programmer uses a template to initialize the driver for each board. The data he must supply, is kept to the bare minimum, and further simplified by extensive use of default values. With the template approach, the programmer can drive a device simply by supplying the basic parameters. He needs to know almost nothing about the inner workings.

The following source code, which performs high-speed acquisition on the DT2821-G, illustrates the simplicity of the template approach. This code sets the DT2821-G for DMA transfer at 200 kHz, which is close to the maximum possible for an AT. Samples will be taken alternately from channel 0 and channel 1, winding up in the array TIME.SERIES. Acquisition will stop automatically when the array is full.

```
\ DECLARE ACQUISITION DEVICE TYPE, ESTABLISH SAMPLING RATE (Hz)
  DT2820
  200000. SAMPLE.RATE :=

\ SET UP THE TEMPLATE
  0 1 A/D.TEMPLATE CHANS_0&1
    1000. SAMPLE.RATE / CONVERSION.DELAY
    TIME.SERIES DMA.TEMPLATE BUFFER
  A/D.INIT

\ ACQUIRE THE DATA
  CHANS_0&1 A/D.IN>ARRAY(DMA)
  BEGIN
    ?DMA.ACTIVE NOT
  UNTIL
```

To initialize the template, it is only necessary to specify the physical device (DT2821), what channel numbers to sample (0 and 1), what sampling rate to use (200kHz), and what array to use as the input data buffer (TIME.SERIES). The template is subsequently referred to by name (CHANS_0&1). The single command, 'A/D.IN>ARRAY(DMA)', starts the acquisition asynchronously to program execution. The while loop, 'BEGIN ... UNTIL', is necessary to make the program wait for the acquisition to terminate.

Diagnosing data problems

Inevitably it is necessary to solve data problems when creating a data acquisition system. The ultimate system product, data, must be examined at each stage during capture, refinement, and presentation. Peripheral devices generate data which holds many clues to where malfunctions have occurred. Fortunately, ASYST has some powerful inbuilt data diagnostic aids. As with the device drivers above, these features form part of the application software which distinguishes ASYST from FORTH.

Arrays are the most prevalent scientific and engineering data structures. Vectors, time series, power spectra, and distributions of all kinds are naturally expressed as arrays. In textbook formulae, arrays are treated as single entities. The textbook says "A+B=C" and, as long as A, B, and C are arrays of comparable type and like dimension, the meaning is clear. The corresponding FORTRAN code is not so simple. Loops are necessary to perform array operations. Worse still, it is necessary to specify the array bounds in the FORTRAN code. The same code will not work on arrays with different dimensionality.

The ASYST treatment of arrays permits most array operations without the burdensome requirement of using the loop structure. Code for adding or multiplying arrays has the same simplicity as code for adding or multiplying scalars. An array is a single entry on the stack. Even more sophisticated array operations such as laminating vectors, taking cross-sections, or locating maximum elements can be accomplished with single commands, rather than loops.

Examining the contents of an array graphically is accomplished with the WORD 'Y.AUTO.PLOT'. This ASYST WORD will autoscale and plot any one-dimensional real or integer array using default settings for all the troublesome details which plotting subroutines generally require the programmer to specify. These defaults can be overridden, but ninety per cent of the time one wants to get a quick look at the data with minimum fuss and bother. The plotting takes seconds. This powerful feature opens a new world of possibility for the programmer and the user.

Summary

ASYST (like FORTH) is a programming environment which greatly enhances the programmer's productivity. This productivity boost is due in part to features derived from FORTH and partly to features unique to ASYST. In the former category, 1) threaded code brings the process of outlining a program's functions in pseudo code very close to the process of writing valid code, 2) the ability to execute subroutines interactively greatly speeds the debugging process, and 3) the stack-oriented operations encourage the use of fewer variables, with consequent reductions in program complexity. In the latter category, 4) a rich set of device drivers and a parametrized device interface simplify interfacing instruments, and 5) data handling is quick and convenient because of the array-oriented mathematical and graphic commands.

"PostScript for the Forth Programmer"

Suzanne W. Hogg
University of Technology, Sydney

Abstract

Postscript "programming" can be indulged in at many levels, from the complete control (but no WYSIWYG) of a fully designed layout written in quite a substantial amount of Forth-like code through to the complete WYSIWYG facilities with the amount of control allowed to the user by the ever-increasing-in versatility software packages which translate directly into PostScript instructions. Artists, architects, engineers, scientists should find the coordinate specifications of the language easy to handle, while the desktop publisher who wants to add a special "something" to the layout should find it rewarding to study the Postscript language at least to the "Cookbook" level.

The ability to design one's own fonts is like the "icing" on the cake.

INTRODUCTION

PostScript for the Forth programmer

The PostScript language is a Forth-like programming language which conveys instructions directly to a printer. It has a wide range of graphic operators and text is handled as though it is a special type of graphic block, thus enabling effects such as printing text in a circular layout to be accomplished with comparative ease.

The user may choose to include PostScript at a varying depth of control

(i)fully programmed blocks of data downloaded straight to the printer

(ii) desktop publishing documents with some more detailed text/graphic manipulations programmed in PostScript.

(iii)word processing documents with special features (logos, diagrams,....) added in, using special fonts which are recognized as direct PostScript instructions.

(i)graphics/desktop publishing programs which themselves convert the

"on-the-screen" imagination of the user into PostScript instructions.

Even if not directly programming in PostScript some knowledge of how the instructions are conveyed to the printer enables the user to more fully appreciate the potential of the software applications using it.

This paper has been produced using a different amount of PostScript involvement on each page.
The opening page is fully programmed and this page(2) has been produced using the blocks/graphics features of a Desktop publishing package, with the footer, header and Title added in by assigning certain blocks to be PostScript. Page 3 is written using a normal word processor, with the footer, header and Title added in by including the PostScript code in a very small fontsize in the font called "PostScript Escape Font", which code does not appear on the printed page but communicates drawing instructions to the printer. Page 4 has exactly the same layout as pages (2) and (3) with another software package doing the entire set of instructions to the printer. Special effects have been imported into page (4) using graphics software packages which translate their pictures into "encapsulated PostScript" for interpretation by the application as printer instructions.

PAGE SIZE and COORDINATES

The A4 letter size is normally printed 8.27 x 11.69 inches. A one-point in the postscript coordinate system is equal to 1/72 of an inch. The coordinates of the "normal" printed A4 page are therefore 841.68 points long and 595.44 points wide.
For simplicity one may choose to design the page as 850 x 600 points - or more conservatively on paper width as 800 x 550.

Some of the (mostly self-explanatory) operators using positioning or simple graphics involving coordinate positions are:

x	y	**translate** so that new origin is x,y.
s_x	s_y	**scale** (scale user space by s_x and s_y)
	θ	**rotate** (rotate user space by q degrees from current origin.
		currentpoint (returns current point coords to x,y)
x	y	**moveto**(shift to the point x,y defined relative to the current origin)
dx	dy	**rmoveto** (shift dx points along x-axis,dy points along y-axis)
		newpath initialize path
		closepath connect subpath back to its starting path

TEXT MANIPULATION

Forth Symposium

Forth Symposium

Φορτη Σψμποσιυμ

PostScript is able to perform great creations with text as objects. It must start, however, with accurate definition of the font (typeface group) to be used.

Because PostScript internally stores its fonts as shape descriptions means that, even when scaled to large, small or unusual amounts, the fonts retain their appearance very accurately.

Three descriptions are required by PostScript

i. what font to find
ii. what size (scaling) is needed
iii. what font is to be regarded as the current font.

Hence we may set three different fonts in which to view the words "Forth Symposium" printed in Postscript at the right of this page.The coding for this display is as follows:-

```
gsave
/displayForthS
{ moveto (Forth Symposium) show} def
530 800 translate
-90 rotate
/Times-Roman findfont 6 scalefont setfont
0 0  displayForthS
/Helvetica-BoldOblique findfont 12 scalefont setfont
0 -20 displayForthS
/Symbol  findfont 24 scalefont setfont
0 -40 displayForthS
grestore
showpage
```

The text of this programming is actually occupying the small "blank space" at the bottom of this printed page, being printed in PostScript Escape font at the smallest available size (9 point).

REPEATED TEXT

One of the exciting features of PostScript comes from graphic creations with text, such as making a flower out of a simple word. To enable repeated transformations of coordinates, shading, sizing etc. it is possible to describe a matrix transformation to be applied to the word or string of words.

FONT CREATION

When preparing a special document one searches for "just the right font for the occasion" - and rarely finds it. As shown already, the font variety is greatly expanded by simple commands specifying scaling in 2 direction, the choice of oblique/bold/italic and the "colour" or grayness of the print. It is
however also possible to create entirely new fonts in three possible ways:-

(i) adjusting a current font
(ii) describing an "analytical font" - defined geometrically, or
(iii) fully describing a new bitmap font.

GETTING HELP
is not easy. The reference books are easy to find - there are not many of them . Some are listed below. The examples are good.

BUT

the slightest mistake may mean you get NOTHING as the output. If doing reasonably complex Postscript Programming/Design it is essential to use the PostScript software/hardware to enable you to run in an interactive mode. If, for example you use the AppleTalk network, which is very easy to connect and run - if the programming is not perfect, you get NOTHING. One longs sometimes for a cheerful error message. Some software does exist to enable "semi-interactive" access to the LaserWriter. Even if subsequently adding postscript code to a higher level workd processing or desktop publishing package it is a good idea to test it out first on an interactive or semi-interactive communication system first.

COOKBOOK TECHNIQUE.

For those who do not want to go to great lengths to develop expertise in programming Postscript, but who would like to use some of the techniques - e.g. putting text vertically or in circular displays for special effects to 'liven up' presentations, the Cookbook technique is recommended - and extremely easy to follow for Forth programmers. Indeed, one of the first and principal texts on Postscript is designed in exactly this way - and disks of Postscript programs are available, enabling the user to make changes to the coordinates, the text to be printed, the radius of curvature, the font..... with little likelihood of upsetting the validity of the Programming itself. A number of Postscript courses are run using this technique and, even for the serious Postscript programmer it seems quite a good way to start.

References

References References

(1) "PostScript Language. Reference Manual" - Adobe Systems Incorporated (Addison-Wesley)(1985)
(2) "PostScript Language. Tutorial and Cookbook"- Adobe Systems Incorporated (Addison-Wesley),1985
(3) "White Spaces" Vol 1 Nos 1,2,3 ,(Graphic Ink, SA),1987-88
(4) "Understanding PostScript Programming" - David A. Holzgang (Sybex)