CROSSTALK Journal of Hewlett-Packard Technical Computer User Groups DEC. 1982

New HP 9000 Computer Family Heralds Entry Into 32-Bit Market

Bill Cummings/ESD

The HP 9000, introduced to customers in mid-November, features models with up to three CPUs, 1 MIPS (million instructions per second) performance, three flexible packaging options, ETHERNET* and HP Shared Resources Manager networking, single - or multi-user capabilities and a choice of HP Enhanced BASIC or a new UNIX**-based operating system.

RESHAPING 32-BIT POWER: THE ASTONISHING NMOS III TECHNOLOGY

HP 9000 performance is equal to other 32-bit super-minicomputers that cost up to four times as much. Its power, small size and relatively low cost are the result of significant HP advances in VLSI circuit technology. This technology, called NMOS III, squeezes a half million transistors onto a single chip — three to eight times the amount currently available in the most dense chips on the market. The resulting circuit spacing is only one micron between devices. Five of these new chips comprise the SPU used in the HP 9000. As a group, they contain the equivalent of more than two million transistors. To put this into better perspective, a standard 32-bit mainframe requires as many as 1000 chips to build a 1 Mbyte system. The HP 9000 uses less than 100 chips.

The HP 9000's "super chips" are mounted on "finstrates" which are specially designed PC boards. These finstrates allow the chip to dissipate its heat directly into the board's copper core where it is drawn off by a small, quiet fan. Because of this unique IC packaging, the HP 9000 can work where the engineer or scientist works — in the office, in the lab or on the factory floor.

PLUG-IN ADDITIONAL CPUs TO MEET FUTURE NEEDS

The HP 9000 features a multiple CPU architecture that lets you configure the computer in ascending levels of performance — the HP 9000 Series 500 with a single CPU, the Series 600 with two CPUs and the Series 700 with three CPUs. In multi-tasking or multi-user environments, adding more CPUs can mean 2X or 3X performance increases at very low incremental cost — without having to rewrite software. The Series 500 and 600 can be upgraded on-site by simply plugging in additional CPU finstrates.

The heart of the HP 9000 is a lunchbox-sized, enclosed card cage (Memory/Processor Module) which houses the finstrate boards containing the CPUs, I/O Processors, 128 Kbyte RAM, memory controller and 18 MHz clock. These components communicate over a Memory Processor Bus with an exceptionally fast 36 Mbyte/sec bandwidth.

Other HP 9000 processor features include a 55 nanosecond CPU microcycle time and a 6 Mbyte/sec I/O rate. Memory cycle time is 110 nanoseconds.

The HP 9000's 32-bit addressing capability enables technical users to easily deal with programs and data structures that are as large as the computer's 500 Mbyte physical memory limit.

FLEXIBLE HP 9000 PACKAGING ALLOWS CUSTOMIZED SYSTEMS

Each series of the HP 9000 is available in three packages: the Model 20 integrated desktop workstation (built-in CRT, keyboard, printer and flexible disc drive), the Model 30 rack-mountable box and the Model 40 offered in a desk-height mini-cabinet.

HP 9000 customers have a choice of two operating systems — a very high performance version of HP's Enhanced BASIC and HP-UX, a fully supported, extended-function version of Bell Labs' UNIX. HP-UX adds several enhancements to the Bell System III UNIX such as virtual memory, improved file reliability, IMAGE Data Base Management and 2D or 3D graphics. It also supports HP Pascal, FORTRAN 77 and C languages.

HP 9000 BASIC is considerably more powerful than standard BASIC systems. It features a new "run time" compiler that provides the friendliness and ease of use common to interpretive BASIC, but adds the faster final execution speeds of a compiled language.

An asynchronous terminal emulator is available for both operating systems.

HP 9000 SOFTWARE TO COME FROM SEVERAL SOURCES

HP plans to introduce several proprietary software packages for the HP 9000 during the coming year, with emphasis on integrated solutions for electrical and mechanical engineering. HPSPICE Circuit Simulation, HP FE II Finite Element pack and HP DESIGN Mechanical Design software are available at introduction.

*ETHERNET is a registered trademark of Xerox Corp.

**UNIX is a registered trademark of Bell Laboratories, Inc.

CONTENTS

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

Superchips keep Computers Shrinking

How small can a computer be? With the arrival of the superchips — complete computer "brains" squeezed onto miniature chips of silicon only slightly thicker than this paper — computers are shrinking even more rapidly than in the past. Powerful computers that once required rooms of their own can now fit into a package about the size of a child's lunch box.

Five of the most remarkable of the superchips have been put to work in a high-performance 32-bit scientific computer, called the HP 9000, just introduced by Hewlett-Packard Company. These quarter-inch-square chips handle all the main functions of the new computer. The largest of the HP superchips contains electronic circuits equivalent to 600,000 transistors, which is up to six times more circuitry than on other commercially available chips.

Together, Hewlett-Packard's five superchips pack the equivalent of more than two million transistors — more than four times the number of parts in a jumbo jet liner.

The chips are able to hold all of these electronics because of an HP advance in very large-scale integrated-circuit technology which squeezes the electronic circuits on the chips down to only one micron, or 40 millionths of an inch apart. To illustrate this circuit density, a pinhead would cover 25,000 of the chips' transistors. With the circuits so close together information gets transferred from one circuit to another 18 million times every second.

The extremely small size of the chips made it possible for Hewlett-Packard to create what is believed to be the world's most powerful computer workstation. Using the superchips compresses the full power of a large, expensive scientific and engineering mainframe computer into the compact HP 9000 integrated workstation. While it is the size of a personal computer, the new HP machine with its power is in an entirely different class. A 32-bit computer typically is used where fast calculations must be made to solve complex mathematical problems. An example of this is computer-aided design, in which products such as automobiles and airplanes are simulated in the form of a math model.

Training Courses

For several years now, the S.E.O. has been providing customer training courses for our customers, including both the HP 1000 and various desktop computers. These courses have typically been designed by the particular division in the U.S.A., occasionally with changes made here to meet our customers needs.

We would like to turn the tables somewhat, and ask our customers what **you want** from H.P. by way of training courses.

If you have any suggestions at all concerning training courses — whom they are directed to, content of courses, structure (self-paced vs. instructor-directed, etc.), time considerations (one week full-time vs. longer-term part-time, evening vs. daytime, etc.), or any other suggestions please let us know.

If we can see a customer need for training which we do not currently provide, we will do everything possible to implement that training.

Please pass comments to:
Philip Greetham,
C/- H.P. Australia,
P.O. Box 36,
EAST DONCASTER, Vic. 3109.

NEW PRODUCTS

SERIES 200 BASIC EXTENSIONS 2.0

The BASIC Extensions has just been released to greatly enhance the already-powerful BASIC language on the Series 200 machines, (previously called 9826/9836).

BASIC Extensions 2.0 includes over 190 keyword additions or extensions to the BASIC 2.0 language system. The command set for BASIC Extensions 2.0 was designed to contribute to the features already available in other HP desktop language sets. These features include advanced I/O such as DMA, fast handshake and interrupt buffer transfers, matrix operations and mass storage enhancements. Other features were added to support new peripherals including the 7908/11/12 Winchester discs and Shared Resource Management.

BASIC Extensions 2.0 is provided as three separate 'soft binary' programs: Advanced Programming, Shared Resource Management, and the Color Video Output. RAM or ROM-based BASIC 2.0 must be resident in the computer before loading any of these binaries. The binary programs can be loaded separately or stacked together to provide optimum memory utilization.

FEATURES OF BASIC EXTENSIONS 2.0

The features can be divided into these categories:-

* Entry and editing enhancements
* Debugging extensions
* Matrix operations
* String utilities
* Timer routines and event controls
* I/O enhancements including DMA and fast handshake
* Buffered I/O capabilities
* Formatting enhancements
* Mass storage enhancements
* Command set 80 disc support
* Callable Pascal or assembly language sub-routines
* BCD (98632A) interface support
* Shared Resource Management support
* Color video graphics extensions for 98627A interface

EXTENSIONS MEMORY REQUIREMENTS

Advanced Programming 150 K bytes RAM.
Shared Resource Management 43 K bytes RAM
Color Video Output 21 K bytes RAM.

New HP 1000 Real-Time Computer Offers Cache Memory, Pipelined Architecture and 3 MIPS Speed

The new A900 technical computer is the highest-performance member of the HP 1000 family, and is believed to be the fastest real-time computer on the market today.

Pipeline technology with cache memory, boosted even more by a fast hardware floating point chip set with scientific and vector instruction sets, provides unmatched computation speed. Three million instructions per second and 560 thousand floating point operations per second provide enough computational power to conquer thousands of applications that have previously been far beyond the reach of minicomputers.

A sizeable 768k bytes of Error Correcting Code (ECC) memory is standard and battery backup is optional to maximize system integrity. The base 768k bytes is expandable in 768k byte increments to a total of 6 Megabytes, providing enough capacity for very large applications.

This powerful new processor is available in a box Computer package (2139A) and in fully-integrated Model 19 Computer Systems (based on the 2199A and 2199B System Processor Units). Standard in all configurations are memory mapping and virtual memory capabilities.

Comprehensive software support includes the RTE-A.1 operating system; programming in FORTRAN 77, Pascal, BASIC, and Macro/1000 Assembly language; Symbolic Debug/1000, DS/1000-IV networking, Image/1000 Data Base Management, and Graphics/1000-II graphics application support libraries.

A full line of HP manufactured interfaces and peripheral devices support the highly flexible configuration of one-vendor systems to satisfy a wide variety of application requirements.

The A900 is particularly well-suited to process monitoring and control, high-speed data acquisition and image and signal-processing applications.

NEW PRICE/PERFORMANCE STANDARD

We believe the A900 has just redefined the price/performance standard for the minicomputer industry," said **Joseph P. Schoendorf, marketing manager** for HP's Data Systems Division.

"It's not only the fastest real-time minicomputer we're aware of — it's one of the most capable. While we dramatically improved our existing HP 1000 architecture to give the A900 its exceptional performance, its software is not only compatible with, but identical to, that of other A-Series computers — the A600 and A700 machines we introduced in February.

With only 25 per cent of the number of parts in our previous HP 1000 F-Series, the A-Series is expected to be the most reliable family of mini-computers ever built by HP."

The A900 processor is implemented in Schottky TTL discrete logic and comes standard with a hardware floating-point processor and HP's Scientific Instruction Set (SIS) and Vector Instruction Set (VIS) firmware. The A900 design makes liberal use of state-of-the-art programmed components and the most advanced Schottky technology available. The floating-point capability is implemented through three LSI chips developed in HP's CMOS/SOS technology. Unlike most processors requiring a separate board for hardware floating point, the A900 floating-point chips are designed as an integral part of the CPU for maximum performance and efficiency. When executing the single-precision Whetstone benchmark (B1), the A900 is capable of nearly 1,200,000 instructions per second. "On a dollar-for-dollar comparison, we know of no other computer that even comes close to this level of performance," Schoendorf added. "Comparably priced computers provide only one-half to one-third, or much less, of the computational horsepower of the A900."

CACHE MEMORY FOR FAST MEMORY ACCESS

Pipeline technology gives the A900 computer an extremely fast 133 nsec cycle time for each successful cache memory access. The cache incorporates hardware address create logic for fast next-address generations and supports a 32-bit data bus to the memory controller. With a hit rate typically 88%, average memory access time is approximately 181 nsec.

HARDWARE FLOATING POINT PROCESSOR FOR FAST FLOATING POINT CALCULATIONS

The standard hardware Floating Point Processor (FPP) in the A900 computer accelerates processing for single and double precision floating point operations to real-time speeds. With single precision, the FPP can do over 600,000 additions and/or subtractions per second, over 500,000 multiplications per second, over 200,000 divisions per second, and over 465,000 conversions per second. Double precision add, subtract, and multiply run at over 250,000 operations per second, divide at over 100,000 operations per second.

NEW PRODUCTS

SCIENTIFIC INSTRUCTION SET FOR SCIENTIFIC AND ENGINEERING CALCULATIONS

A Scientific Instruction Set (SIS) of nine single precision and nine double precision trigonometric and transcendental functions and a polynomial evaluation instruction uses the fast floating point computational power of the FPP to solve complex scientific and engineering calculations quickly and accurately.

VECTOR INSTRUCTION SET

The Vector Instruction Set (VIS) applies the floating point processing power of the FPP to highly-efficient repetitive processing of vectors and matrices. Because they take advantage of the inherent efficiency of vector processing, the VIS instructions can achieve extremely fast rates.

HIGH-LEVEL PROGRAM ACCELERATOR INSTRUCTIONS

The A900 base set includes instructions designed to accelerate the execution speed of programs written in FORTRAN or Pascal. These routines speed up parameter passing and other commonly used high-level program operations 2 to 20 times, compared to the same routines in software.

VIRTUAL CONTROL PANEL

A ROM-based Virtual Control Panel (VCP) program enables an operator to perform control panel functions via a local or remote-connected terminal or an adjacent HP 1000 Computer System through a standard serial or DS/1000-IV I/O interface card. Only one I/O card in the system can be given this capability at any one time. That I/O card can connect to a terminal or other computer system accessible only to the system manager or the maintenance department. The operator at the VCP terminal or system can examine and change the contents of registers and memory locations, initiate the macrocoded self test, and select a bootstrap loader and initiate the boot-up of a system.

Because of its remote operating ability, the VCP can be used for remote isolation of system faults, which can help to minimize support costs for OEM products that use A900 components. When not being used as the VCP, the VCP-assigned terminal can be used in the same way as any other terminal connected to the system.

BOOT-UP SOURCES AND AUTO BOOT-UP

The A900 computer supports boot-up from any of the following sources, the first three of which can be used for auto boot-up.

1. An adjacent HP 1000 System in a DS/1000-IV network via the 12007 A/B or 12044A interface.
2. A disc memory via the 12009A HP-IB interface.
3. 12008A PROM Storage Module.
4. Cartridge tape unit of CS/80 fixed disc via the 12009A HP-IB interface.

MEMORY SYSTEM

The memory system for A900 is a dynamically mapped array with base memory of 768k bytes of Error Correcting Code (ECC) memory expandable to 6 Megabytes in 768k byte increments, by adding up to seven memory array cards. Standard ECC assures system availability and reliability. The ECC system detects and corrects all single-bit memory errors and detects all double-bit errors. The ECC system also retains error syndromes when a correction occurs.

Memory access is managed by the dynamic mapping system, a powerful combination of hardware and special instructions. Because of this built-in capability, A900 users can efficiently use large memory systems with minimal programming effort.

INPUT/OUTPUT — DISTRIBUTED INTELLIGENCE ARCHITECTURE BOOSTS I/O EFFICIENCY AND SIMPLIFIES PROGRAMMING

Computation and input/output are often both controlled by the central processor. In the A900 system, the central processor has been relieved of I/O DMA processing. That function has instead been assigned to an individual I/O processor (IOP) on each interface card. Thus, the CPU is free to do its real job of processing data. The CPU, the IOPs on each interface, and memory all communicate with each other via a common bus.

LOW-OVERHEAD I/O

I/O Processor-Managed DMA. The built-in intelligence of each IOP supports autonomous control of I/O operations. This includes high-speed direct memory access (DMA), and can even include chained multiple DMA transfers without interruption of the CPU except at the start and completion of the entire chain.

Simplified I/O Programming. The same level of intelligence that supports DMA-per-channel operation also simplifies I/O programming. The master IOP logic provides for recognition of interface I/O addressing independently of I/O card position on the card cage bus. This lets you standardize I/O addresses and functions in programs without requiring any particular arrangement of I/O cards in the card cage.

COMMUNICATIONS SUPPORT

The A900 processor can communicate with terminals and other systems in the following ways:

- With terminals via single-channel interface or eight-channel multiplexer.
- With other HP 1000 systems via DS/1000-IV HDLC point-to-point interface or multidrop Data Link.
- With HP 3000 systems via DS/1000-IV Bisync point-to-point interface.
- With HP 1000, HP 3000, and other systems via DSN/X.25 interface to packet-switching networks.

COMPATIBILITY WITH OTHER HP 1000 COMPUTERS

The A900 computer instructions are identical with HP 1000 A600 and A700 instructions. The A900 executes the same HP 1000 base instruction set (arithmetic, memory reference, and register reference instructions) as HP 1000 L, M, E and F-Series Computers. Except for dynamic mapping instructions, virtual memory instructions, and I/O instructions, other instructions beyond the base set as defined have the same mnemonics and format as they do in HP 1000 M, E, and F-Series Computers, which facilitates program transport ability between HP 1000 A900 Computer and other HP 1000 Computers.

I/O drivers written for use with RTE-L/XL operating system will run without change in an A900 computer operating under RTE-A.1 provided CPU timing is not a factor. Drivers written for use with RTE-IVB or RTE-6/VM will have to be rewritten for use on RTE-A.1.

KEY FEATURES OF THE A900 INCLUDE:

- Extremely high performance Computer.
- Built-in hardware floating point processor with scientific and vector instruction set firmware.
- 768kb ECC memory for maximum system integrity, expandable to 6Mb.
- Pipeline architecture and 4k byte cache memory for fast memory access.
- Distributed intelligence I/O with DMA per channel for I/O efficiency.
- 50% CPU performance is available at 3Mb/sec I/O bandwidth.
- Compatibility with all A/L-Series I/O interface cards.
- 20-slot card cage with 15 slots in 2139A, 13 in 2199 A/B available for memory expansion and I/O interfaces.
- RTE-A.1 Real-Time Executive operating system, optional with 2139A, included in 2199 A/B, that supports:
  - — Multiprogramming and multi-tasking facilitated by dynamic memory partitioning.
  - — Up to 255 user partitions, which may be reserved and/or allocated from a dynamic memory pool.
  - — Virtual memory for data arrays up to 12.6M bytes divided between main memory and fixed disc.
  - — Up to 15 Extended Memory Areas for data arrays up to 2 Megabytes per program, each sharable by up to 63 programs.
  - — Program development in FORTRAN 77, Pascal, BASIC, and Macro/1000 Assembly language.
  - — Image/1000 data base management, Graphics/1000-II graphics software, and distributed systems networking.
- High reliability and maintainability through the use of simple packaging, self-test, and board level diagnostics.
- Instruction and program compatibility with other members of the HP 1000 family protects software investment of current OEMs and end users and gives users access to a broad base of proven software.
- Built-in dynamic mapping system, memory protect, and time base generator.
- Power fail detection and auto restart with optional battery backup for up to 3 Megabytes of memory.
- Fast on-line system generation.
- Boot loaders included for boot-up from:
  - — Other computer system in a DS/1000-IV Distributed Systems Network.
  - — Disc drive or its integral cartridge tape unit.
  - — PROM Storage Module.
- Remote loading and diagnosis for programming and operation of systems from remote sites.
- Designed to comply with UL, CSA, and IEC safety standards and VDE and FCC RFI standards.

New H.P.D.C.U.G.V. Venue

The November meeting of the group was held in the excellent conference facilities at the C.S.I.R.O. Division of Geomechanics in Glen Waverley. Situated at the end of a quiet suburban street in a gum tree setting the facility was most conducive to 'gettin away from it all' and concentrating on the meeting's events.

The meeting got off to a lively start with a spirited presentation by Barry Liston of Assco on the new H.P. 86. On display were the H.P. 86, both 9" and 12" monitors and the low cost 9130A disc drive. Barry received an excellent response from the audience who were obviously very interested in this product.

We were then treated to a resume of forthcoming H.P. products by Stan Karpowicz of H.P. Stan spoke about renumbering of the computer products into new families and H.P.'s intentions to move toward more standardisation of operating systems, languages and file structures. Members were particularly interested in the new 3½ inch H.P. diskette also mentioned.

Tony Stevens then took the floor to review mass storage systems available to H.P. desktop computer owners. With well prepared displays, Tony illustrated the range of H.P. and non-H.P. mass storage devices suitable for each of the popular desk-top computer products.

Following a vote of thanks to Rory Cox of C.S.I.R.O., who had organised the venue, the meeting concluded with coffee and biscuits and general informal discussion.

B. T. O'SHANNASSY

H.P.D.C. U.G.V. Plans for 1983

Our first meeting for 1983 will be held at C.S.I.R.O. Glen Waverley (end of Kinnoull Grove) on Tuesday, March 1 at 4 pm. The main topic entitled "Strings and Things" will be presented jointly by Chris Simpson and Ian McWilliam. Chris will discuss handling Strings on computers from 9825 up, and Ian will explain how our own membership list is compiled and maintained.

Our financial year finishes on February 28 and subscriptions fall due then. Remember that "CROSSTALK" will be mailed to financial members only.

The next meeting (tentatively scheduled for mid April) will be our Annual General Meeting. The final date will be timed to fit in with a whizz-bang product display yet to be announced by H.P. We hope to have several overseas speakers for that meeting.

Further tentative meeting dates are:
7th July, 6th September, 9th November.

Topics presently being considered for those meetings are:

CP/M on H.P. Desktop Computers
H.P. Graphics
Mastering the H.P. 85.

We are also hopeful of organising presentations by software vendors.

If there is some other subject you would like to see covered, don't keep it to yourself — call a committee member and air your views!

Short talks on applications of desktop computers are always popular at our meetings. How about volunteering to talk about YOUR application?

B. T. O'SHANNASSY

PLOTTING ONTO PRINTERS

KEYWORDS: desktop computers, dot-matrix, graphics, plotting, printers, raster.

"I've a 9825 and want to plot to our 9876 'graphics' thermal printer and, while you're at it, also to our 1350A vector-graphics display."
"Er, yes . . . but can't you get hold of a 9872 plotter or something else that speaks HP-GL? It would save lots of fiddling about!"
"But they all DO have HP-IB interfaces . . !"

OK, so it happens. Quite often. Its not always the salesman's fault that one is stranded with subtlely incompatible equipment. And it is not always possible to buy the easy way out, what with HP hardware prices.

It's only the super new desktops that have finally got the plotting act together. They can cope with nearly all HP 'graphics' devices by allowing a simple declaration that the "PLOTTER IS Whizbang II". Then the normal plotting statements can be used. 'Unified' plotting, you understand.

Not all 9845s can do this trick, though. Nor can the 9835 or the brilliant (and much maligned) 9825. Series 80 also has its limitations. High-level plotting onto 'graphics' printers such as 9876A, 2631G or 2731G is simply not possible.

But wait! A 9845 with graphics or Series 80 can plot to its screen using normal plot statements. Then the screen can be DUMPED dot for dot onto a graphics printer (inbuilt or otherwise). Great! But if your computer cannot DUMP, read on . . .

The dot-matrix 'graphics' printers in question have ONLY ONE graphics command: "Print-a-line-of-dots". Somewhat rudimentary, I agree, but we can use it. On any computer. To print a replica of a graph, we must effectively draw the graph in a big rectangular field of dots. A raster. Just turn on the dots closest to the line of the graph. Leave the others off. Then starting with the top row of the rectangular raster, Print-the-line-of-dots for each row in turn onto the printer, and lo, we have a printed graph. Easy?

A major problem exists: a LARGE chunk of memory is needed to store this big rectangular raster. Each dot can be represented by one bit of memory, as it can only be on or off. So, eight dots would require one byte of memory, that is, one character of a string variable. A 9876A thermal line-printer uses 560 dots per row, so to print a full-width raster line would therefore require 560/8 = 70 bytes (720/8 = 90 for a 2731G). Vertical spacing of rows for the 9876 is 77 per inch (90 per inch for a 2731G). Thus a full width raster five inches deep would use 70 × 77 × 5 = 26,950 bytes (40,500 for a 2731G).

If memory is a limitation, reduced raster size is one answer.

A simpler answer is valid in the case of serial single and multiple plots, as we shall soon see. For some other plots, however, Y values at one X value may be influenced by events at other X values. For hidden-line plots, whether or not to turn on dots depends on the other curves too. For multiple curves, all the data may not fit into memory at once. In other words, there are cases when the whole picture needs to be built up piecemeal in memory, hence requiring a full raster. (And another article).

A GOOD, SIMPLE APPROACH is to turn the whole plot on its SIDE so that the X-axis runs DOWN the page. Each successive row now represents successive values of X. In fact, we can calculate the Y value for that row, turn on the correct dot and print the dot row immediately. Then the next row of dots may be prepared for the next X value, etc. A string variable of only 70 (90) characters is needed — it is used over and over again. What is more, the graph could be sixteen feet long!

Some likely questions about this method follow:

1. WHICH DOT IS TO BE TURNED ON? — If dot no. 1 (the left most dot) represents Y min and dot no. 560 represents Y max,

then for any value of Y, we turn on dot no.:
D = 560* (Y-Ymin)/(Ymax-Ymin) - or better,
D = max (1,min(560,int(.5+560*(Y-Ymin)/(Ymax-Ymin)))) where D is guaranteed to be in the range 1<=D<=560.

2. HOW DO WE TURN ON DOT NO. D? —

a) Reset the single line raster string R$ to all char (0)s.
b) The character number, C = int ((D+7)/8) (between 1 and 70)
c) The Bit No. (within the char), B = 8*C-D (between 0 and 7)
d) Set the raster-string character:
R$[C,C] = char (2^B) or,
R$[C,C] = char (ior((2^B), num(R$[C,C])) for multi-curve plots, to retain any dots already set.

WHAT IS THE PRINT COMMAND? —
If a 70-byte string R$ represents a single row of dots, and P is the printer select code:

wtb P, char (27) &"*b70W"&R$ for 9825, or PRINT USING "#,k"; chr$ (27) &"*b70W"&R$ for 9835/45.

Note that the carriage-return-line-feed is suppressed.

HOW CAN THE X-AXIS BE DRAWN? —
Determine which dot number on the raster row represents Y = 0 and turn this dot on as well before each row is printed.

AND THE Y-AXIS? —
Include in the print loop an IF statement that detects the occasion when X = 0. In this case, simply turn ALL the dots on. (Each character in R$ is set to char (255)).

CAN THE AXES BE LABELLED? —
Yes. Each time (horizontal) text is to appear, the required text line is printed, correctly formatted. But the line-feed is suppressed and the carriage-return sent alone. This has the effect of printing and rolling the paper back one line (or simply not advancing the paper). Typical print statements are:

wtb P,T$,13 (where T$ contains the text), or
PRINT USING "+,k"; T$

JOINING DOTS —
Often successive Y values will be more than one dot removed horizontally from each other, especially in wildly fluctuating graphs. In this case it is desirable to fill in the line between Y-values, by turning on the closest intervening dots. Variously complicated algorithms can be used. But in the simplest case, where there is a Y value calculated for EACH row of dots, one can turn on all the dots from the (previous Y value + 1) up to the new Y value and then print the row.

Note:
1. Dot-fill techniques are worth a whole article!
2. To SHARE dots with the previous line would seem more symmetrical, but requires looking ahead to the next value before printing the current row.
3. For widely separated X-values, a more complex method is necessary.

Perhaps some readers will recall the bad old days before computer graphics was invented. Lovely pseudo-curves were obtained using identical methodology to that described above. Still valid is this 9830/9866 example of plotting a SIN curve:

```
10 FOR X=1 TO 2*PI STEP PI/20
20 WRITE (15,*)
     TAB(79*(1+SIN(X)));"*"
30 NEXT X
```

Horizontal resolution of the resulting 'asterisk' graph is one character (0.1 inch) rather than one dot (about 0.01). But essentially, nothing much has changed!

Chris R. Simpson,
Simpson Computer Services P/L
(03) 859 6643

Dumping Graphics on the 9826/36

The 9826/36 has two built-in graphics dumping routines, invoked by simply declaring:

DUMP DEVICE IS 701
DUMP DEVICE IS 701, EXPANDED

The first gives a straight raster dump to the system printer 110 × 145mm (for the 9836) left-justified on the page. The latter gives a factor of 2 enlargement, rotated 90 deg. on the paper.

I was recently posed the question:
How can we dump the graphics to be right-justified on the paper, rather than the standard left-justification?

The following program was written for the 9826/36 to do just that. The main program allows you to choose a margin to suit your needs and positions the graphics dump accordingly, anywhere between the left and right margin of the paper.

```
100   ! RE-SAVE "DUMP_GRAPH"
110   ! Program developed to position the dumped graphics screen
120   ! Philip Greetham & Robert Dey S.E.O. Melbourne DATE: 18 October 1982
130   !
140   OPTION BASE 1
150   GCLEAR
160   GRAPHICS ON
170   INTEGER Picture_no,Offset,Graphics_width,Paper_width
180   !
190   STATUS 1,9;Graphics_width                    ! Determine if 9826 or 36
200   !
210   ASSIGN @Picture TO "PICTURE2"                ! 'GSTORED' picture file
220   !
230   IF Graphics_width=50 THEN                    ! For 9826
240     ALLOCATE INTEGER Screen(300,25)
250   ELSE                                         ! Has to be 9836
260     ALLOCATE INTEGER Screen(390,32)
270   END IF
280   ENTER @Picture;Screen(*)
290   GLOAD Screen(*)
300   !
310   INPUT "Please enter the required left margin:",Offset
320   Dump_graphics(Screen(*),Graphics_width,Offset)
330   !
340   ASSIGN @Picture TO *
350   END
360   !
370 Dump_graph:     SUB Dump_graphics(INTEGER Picture(*),INTEGER Width,Offset)
380                 ! Contents of the graphics screen passed into Picture array
390                 !
400                 OPTION BASE 1
410                 ! Data$ is the 'massaged' graphics data
420                 DIM Data$(400),Fill$[26]
430                 !
440                 ASSIGN @Printer TO 701
450                 INTEGER Row,Column
460                 !
470                 IF Width=80 THEN Width=64       ! 9836 graphics width
480                 IF Offset<0 THEN Offset=0       ! Check range of margin
490                 IF Offset+Width>90 THEN Offset=90-Width
500                 !
510                 ! This escape code sequence sets graphics mode in the 26716
520                 ! and tells it to expect width plus offset binary bytes
530                 ! of information
540                 Esc$=CHR$(27)&"*b"&VAL$(Width+Offset)&"W"
550                 !
560                 ! This is the blank pad to be output on each line to move
570                 ! the picture over to the right side
580                 FOR Fill_chr=1 TO Offset
590                   Fill$=Fill$&CHR$(0)           ! Offset chars are nulls!
600                 NEXT Fill_chr
610                 !
620                 OUTPUT @Printer;CHR$(27)&"*rA" ! Start raster mode transfer
630                 !
640                 ! Now build the output string to create the picture
650                 ! row by row
660                 FOR Row=1 TO 390
670                   Data$=""
680                   FOR Column=1 TO 32
690                     Upper_byte=(SHIFT(Picture(Row,Column),8))
700                     Lower_byte=BINAND(Picture(Row,Column),255)
710                     Data$[LEN(Data$)+1]=CHR$(Upper_byte)&CHR$(Lower_byte)
720                   NEXT Column
730                   Data$=Esc$&Fill$&Data$        ! Final output string
740                   OUTPUT @Printer USING "K";Data$
750                 NEXT Row
760                   OUTPUT @Printer;CHR$(27)&"*rB"  ! Flushes Graphics Buffer
770                   OUTPUT @Printer;CHR$(13),CHR$(10)
780                   OUTPUT @Printer;CHR$(13),CHR$(10)
790                 SUBEND
800 End_dump_graph:             !
```

Series 9800 Desktop Computers Exchange Library Software Catalog

Hewlett Packard market this volume four times a year. The catalog contains brief descriptions of programs that are available for exchange. If you submit a program that is accepted for inclusion in the next Catalog then you are entitled to make a free selection of 3 programs from the Library. For further information on programs in the catalog ring the HPDCUGV Librarian.

SERIES 9800, 1000 AND 80 SOFTWARE CATALOG

Hewlett Packard market 4 times a year a 'Technical Computers HP and HP+ Software Catalog' for use by program writers. This book contains the description of various programs in the fields of Engineering, Mathematics, Management and Communications. Some of the binary utilities available, and useful on 9845 systems are:

NODUP — prevent duplication of program tapes.
FIND — provide a global search of your program for a literal string.
DUMP ALPHA — enables a CRT dump to the current printer.
XREF — variable, label cross reference lister.
STRIP — comment stripper.
LOAD SUB — enable STORED sub-programs to be LOADED into memory at the bottom of the current program.
:E — defines blocks of main memory to act as a mass storage device.

For further information on these or others in the Catalog ring the HPDCUGV Librarian.

BINARY PROGRAM SNIPPET

Many programs that are purchased as 'LOAD' programs contain inbuilt binary routines. These can be stripped by first loading the program and then executing a STOREBIN command. This will give an error if no binary programs are in memory. An example of its use is the extraction of the 'ON KBD' binary used in programs with menu selection. The stored binary can then be used for other programmes that require ON KBD statements inside sub-programs.

H.P.D.C.U.G.V.

If any members have changed their names, addresses or telephone numbers, would they please notify our membership secretary, Ian McWilliam, on 819 8864.

Focus 1000

DEBUG/1000

John D. Johnson,
Data Systems Division, Hewlett-Packard Company.
July, 1982.

1. ABSTRACT

Debug/1000 is a full-capability, interactive, source level symbolic debugger for HP 1000 computer systems. The philosophy behind Debug is that programmers should be able to examine and modify the dynamic behavior of their programs in the same terms they use in creating them. Also, the debugger should not intrude into the user program or affect its performance in any way. These design goals are achieved by enhancing the compilers and linker to provide the required symbol table information and by extending the operating system to support program monitoring from a separate partition. Breakpoints are implemented by replacing a user instruction with an operating system trap instruction. Creative utilization of these features results in a very powerful program debugging tool.

KEY WORDS: Debugging, Program development, Interactive tools, Symbolic debugging.

2. INTRODUCTION

Debug/1000 is a full-capability, interactive, source level symbolic debugger for HP 1000 systems. It allows programmers to examine their programs in terms of the language in which they were written. By using a full screen CRT display, users can interactively follow the flow of their programs while viewing the original source code. They may display and modify program variables using the same names and formats that their program uses. Single integer through double complex, logical and character data types are all supported. Breakpoints allow programs to execute at full speed until the area of interest is reached. Then source level single stepping and variable tracing facilities allow a detailed examination of program behavior. Conditional breakpoints can be used to stop programs when a specific condition is reached.

Friendly human interaction is a major feature of Debug/1000.

A simple but powerful command set is quickly learned by programmers. English error messages and an online "help" facility allow even novice users to perform sophisticated program examinations. A "command stack" allows recently entered commands to be edited and re-executed, thereby greatly reducing the number of keystrokes required by the user. This stack is even saved across invocations of Debug so that a user can easily return a program back to the state of interest.

Users communicate with Debug in terms of variable names, source line numbers and procedure (subroutine or function) names, An example of what the user sees during a debugging session is presented in Figure 1. To list subroutine FFT, the user simply types 'L FFT'. Debug uses its symbol table to find the file and line number in which FFT resides and then displays the first few executable lines on the screen. To set a breakpoint at FFT, the user types 'B FFT' and Debug will insert a trap instruction at the appropriate address and report the line number and module name to the user. Table 1 presents a command summary.

How these features are implemented is the main subject of this paper. First we examine Debug's philosophy on program debugging, then determine the system requirements to achieve the desired goals. How the various system programs achieve these requirements is presented next. Finally, how Debug accomplishes some of its feats is presented.

B <Location>	Sets a breakpoint at location specified.
C <Location>	Clears the breakpoint set at location specified.
D <Locations>	Displays variable.
E	Aborts your program and exits Debug.
G <Location>	Allows your program to proceed from location specified.
I <f1 [f2]>	Executes a set of commands from a file (f1) and optionally logs the output to f2.
L <Location>	Lists a screenful of source code in your program.
M <Loc><value>	Modifies the value of variable.
P <line>	Allows your program to proceed to the next breakpoint or specified line.
S	Steps to the next line of source code.
T <Location>	Shows location executed without stopping program.
V <number>	Changes the number of source lines displayed on screen.
W	Shows the callers of the current subroutine.

Table 1. Debug/1000 Command Summary

```
   40          IM  = SIN (ANG)
   41    2     IF ( .NOT. NEW .AND. K*K0 .GE. 1 ) GO TO 4
   42 C      ---------------------------------------
   43 C        COMPUTE TWIDDLES IF NECESSARY ...
   44 C      ---------------------------------------
   45          U(1) = CMPLX ( RE , -SIGN(IM,FLOAT(K)) )
   46          DO 3 I = 2,L2N
>  47    3     U(I) = U(I-1)*U(I-1)
   48          K0 = K
   49 C      ---------------
   50 C        BUTTERFLIES.
   51 C      ---------------
   52    4     SBY2 = N
   53          DO 7 STAGE = 1,L2N
   54          V = U(STAGE)

DEBUG> b 47/fft
Breakpoint set at 47/FFT


DEBUG> p
Break at 47/FFT
DEBUG> d L2N new re u(1)
L2N = 5      NEW = true      RE = 0.980785
U(1) = (0.980785,-0.19509)
DEBUG> m L2N 6
L2N: 5 => 6
DEBUG>
```

Figure 1. Typical Debug Session

3. DEBUG'S PHILOSOPHY

Major difficulties in finding errors in programs often occur because the programmer is trying to work from inadequate evidence. The actual flow followed by a program is not recorded anywhere, nor is the history of a variable's values. Without an interactive debugging tool, programmers have only limited methods for determining the dynamic behavior of a program. Experienced programmers rely on their intuition to determine where and how many print statements should be inserted in order to gain sufficient clues for understanding program behavior. This is a slow and iterative process. Also, modifications can not be tested without re-editing the source, recompiling, relinking and re-executing the program. To overcome these obstacles, two fundamental capabilities of Debug/1000 are explicit control over a program's execution flow and explicit control over a program's data state.

Information is nearly useless if it is presented in a form difficult for a human to interpret. Octal dumps, one of the traditional debugging methods, present much extraneous information and are extremely crude. The major disadvantages of most debugging techniques include: not presenting the information in terms of the source language, presenting much irrelevant detail and presenting the data in a different format than the one the programmer thinks in.

These deficiencies are overcome by a symbolic debugger. It must understand the symbols used in the source code and it must be able to convert the data types supported by the machine into representations used by the programmer. There must also be a convenient means to override the declared representation because many times the data stored in a variable is not the same as the declared type.

To successfully debug, programmers must analyze both program execution flow and data state simultaneously. They need to be able to concurrently view both the section of interest in the source code and the values of selected variables in order to easily grasp the condition of the program under test. Traditionally this is done using clumsy listings for examining the source and a terminal for examining the data. The major drawback of this technique is the time and expense required to make a listing. Listings quickly become incorrect as changes are made to correct bugs. Debug/1000 overcomes the need for listings by presenting the user with a split-screen display. The top half of the CRT display is used to present the source text. Whenever a breakpoint is encountered, a portion of the original source file around the breakpoint is displayed. The user can then look at other parts of the program or even at other files. The lower half of the screen is where the user inputs commands to Debug and where the current values of variables are displayed. Thus, it is possible to do 'paperless debugging' which is ecological but, more importantly, improves programmer productivity.

A prime requirement for Debug/1000 is that it must not affect program behavior. It is not acceptable for special code to be required in the user's program. There should be no need to restructure a program just to debug it. The production software should be identical to the final development software. This means that no user code space is lost and no extra statements must be added in order to debug. The program being debugged must run exactly the same as it would when not being debugged. No bugs can be introduced by the debugger, and no bugs can disappear when the debugger is present, only to reappear when the debugger is not used. All this is accomplished by running the debugger in a separate memory partition from the program being debugged.

There are many advantages of putting Debug in a separate partition. As just stated, this allows Debug to be non-intrusive.

Debug does not affect a program in any way when it is not being debugged. The separate partition also protects Debug from any uncontrolled, destructive writes by the user's program. No matter what the user's program does, Debug can still examine it. Because Debug does not use any of the user's address space, it can be relatively large and powerful.

A final requirement is that Debug must be easy to use. It must be fast and friendly. It should tell the user what it is doing and allow him to interrogate the current state for breakpoints and other options. English diagnostic messages and a online help facility should provide a human-oriented interface. A few simple but powerful commands minimize the number of keystrokes required, but not at the expense of cryptic input sequences. Debug's understanding of the scope of names used by the program enables it to provide appropriate defaults. The number of keystrokes required are also reduced by saving all commands in a command stack. This stack is saved between runs of Debug since it is likely that you will run a program more than once before finding the exact area needing correction. The commands in the stack can be edited and repeated as required.

4. SYSTEM REQUIREMENTS

Debug must have explicit control over both the execution flow and data state of a program. These capabilities are provided by the operating system but only in a primitive form. Debug translates the primitive information into the representation the user is familiar with. Execution control is achieved with the use of code breakpoints. This is the process of replacing a user instruction with a special trap instruction which transfers control to the debugger. Debug/1000 uses a HALT instruction for this purpose. This is an illegal instruction when the system is operating in user mode and it causes a violation when it is executed. The operating system traps this and, if the program is being debugged, transfers control to the debugger.

Correct handling of violations by the operating system is not a trivial matter. Normally when a user's program causes a violation it is aborted. Its state is destroyed and a rude message is printed to a terminal. However, if breakpoints are implemented using illegal instructions then the operating system must perform special processing upon the occurrence of a violation in the user's program. All of the user's state information must be carefully saved and no messages should be issued. To inform the operating system what action to take upon violations, a "debug bit" was added to the operating system's "ID segment" tables. When this bit is clear then normal abort processing occurs upon a violation. When this bit is set then a violation causes the offending program to be suspended in the same manner as when the operating system pre-empts it. Debug is then scheduled and is able to examine the suspended program's image. It has the responsibility of determining if the violation is due to a breakpoint Debug inserted or if it is due to a real violation by the program being debugged. This method of giving Debug the responsibility for all violations while a program is being debugged has the advantage of allowing the user to examine the corpse of his offending program and possibly even correcting the error and continuing.

Memory overlays by program segments must be communicated to the debugger. When there is a breakpoint in the overlay area it disappears when an overlay occurs. When the original segment is brought back, this breakpoint must be re-installed. To handle this properly, whenever a segment overlay is requested and the debug bit is set, Debug is scheduled and given access to information concerning which segment has been brought in. Debug must maintain tables describing which segment is currently loaded and which breakpoints must be re-installed. Debug can re-install breakpoints in segments because it receives controls after the segment is loaded but before the user program is allowed to execute any instructions in the new segment.

Control over a program's data state is achieved by giving the

debugger access to the address space of the program being debugged. If the debugger resides in the same address space as the program being debugged, this is easy. However, Debug/1000 resides in a different address space, so there must be a method for mapping at least a part of the program being debugged into the address space of the debugger. This is supported under the RTE operating system by memory locking the program being debugged so that it will not be moved to a different physical memory address. The debugger is allowed to use the physical memory mapping instructions when accessing the target program. It uses the operating system's tables to find the physical page corresponding to the logical absolute address it wishes to reference. Debug then maps this page and the following page into its high two logical pages and copies the data to or from a local buffer. Two pages are mapped to insure proper operation when the required data falls across a page boundary.

5. SYMBOL TABLE REQUIREMENT

Debug/1000 is a symbolic debugger. Compilers must communicate their symbol table information to Debug. A dump of the compiler symbol table alone is insufficient because the addresses in it are all relative to the base of the module being compiled. To examine and modify variables, Debug must know their absolute addresses and this information can only be provided by the linker. Thus, both the compilers and the linker had to be extensively enhanced to support Debug.

Compilers now communicate the required information to Debug via additional relocatable records. Obviously, there is a symbol record. It contains the names of the symbols, their data types, and their relocatable addresses. If a symbol is a parameter then its parameter number is included. If a symbol is an array then information about the number of dimensions, size of each dimension, and the lower bound of each dimension is passed. There are many more special cases, such as character string descriptor. Debug needs to know how to use a string descriptor to access the actual run time characters.

Line number records provide the information which enables Debug to map between addresses and source line numbers. To conserve space, this information is encoded into blocks. Each block begins with a starting address and a corresponding line number. Following in the rest of the block is the number of code words required by successive lines. Comments generate zero words of code and the compiler decides if it would be smaller to emit several zeros or to start a new block. The record format also supports more than one statement per source line.

Module name records were extended to include the source file name from which the code was compiled. This allows Debug to use the original source file when displaying the program as it executes. Entry point records were extended to supply starting line and calling sequence information. Compilers may generate procedure prologue code containing a jump to the actual starting line. When the user sets a breakpoint at a procedure, he expects that this prologue will be executed before his breakpoint is reached. Debug uses the starting line information from the entry point record to determine the proper address for this type of breakpoint.

Calling sequence information is needed because rapid single stepping over a procedure requires that its return address be known. This information is especially valuable when a variable number of parameters are passed. Luckily, the standard calling sequence supports a variable number of parameters quite well. If Debug knows that the standard calling sequence is being used then it can easily find the return address. If a procedure has more than one return point then Debug is informed of this so that it can set multiple breakpoints and discover the actual run time return point used.

Converting the relative addresses to the absolute addresses is the job of the linker. It combines the debug information from all the relocatable files used during linking of the program into one file which is used by Debug. This file has all the modules, entry points, symbols, and line numbers as well as their absolute addresses. Information about segment overlays is created by the linker and supplied in this file.

All these debug records can cause the total symbol table to become very large. Debug manages this large amount of data by reading in the file supplied by the linker and arranges the symbols into several data structures which allow efficient searching. Modules, entry points, and local variables and arranged into binary trees.

Linked lists are woven through the binary trees to allow the accessing of parameters and other types of special information. Line number information is packed close together to reduce the amount of disk paging required and is linked to its module. All the tables are required because it must be easy to map from a symbol, entry point or line number to an address and vice versa.

6. DEBUG'S ALGORITHMS

Most of Debug's feats are straightforward once it has built its symbols and has access to the target program memory image. To display a variable, Debug finds its name in the symbol table, gets its address, reads its value from the target memory image and formats the data according to its type from the symbol table. To set a breakpoint, Debug consults the entry point or line number tables to find the address, retrieves and saves the old value at this address, and pokes a HALT breakpoint into the target program. Care must be taken not to confuse the saved value with a breakpoint already at the requested location. Listing the source once a breakpoint is reached is achieved by obtaining the address at which the breakpoint occurred and mapping it to a module and line number. Once this is known, the file name is extracted from the symbol table and the screen display routines are called. They read from the source file and write to the top half of the screen where the source text is displayed. To decrease update time, the display routines keep track of which lines are currently displayed. If the new screen is only a few lines above or below the ones currently displayed then the appropriate number of lines are deleted and the new lines inserted. Update time is reduced because most of the lines just change position and are not redrawn.

More sophistication is required to execute a single step command. The normal single step is to step one source line. This is achieved by examining each instruction in the line of code being stepped over and setting breakpoints at all the possible locations to which line can transfer. The user's program is then allowed to execute until it reaches one of these breakpoints. Debug regains control and clears the breakpoints it just installed. The simplest case is when every instruction in the line to be stepped over has only one destination. In this case, only one breakpoint is needed to regain control after the execution of the line. Things are more complicated if there are conditional jumps in the line. Debug determines the destination of each jump and sets a breakpoint there if it is outside the line being stepped over. Destinations inside the line are traced until their path extends beyond the line being single stepped.

Single-stepping over a line becomes very complicated when there is a subroutine call in it. Debug is designed to step over the subroutine and to do this it must know the return address. On the HP-1000, subroutine parameters are usually passed in line by pointers following the subroutine call. Debug can not distinguish these pointers from instructions. Also, a subroutine may have more than one return point, so Debug may be required to set more than one breakpoint. Debug tries three strategies to determine what breakpoints are required. First, it checks if the compiler supplied information describes the number of parameters and the number of return addresses. If this information is provided then it is used for setting breakpoints. Otherwise, Debug must determine the return addresses on its own. It examines the code to be executed in an attempt to ascertain that the standard calling sequence is used. If it finds this to be the case, then the return address can be determined by the first data word after the subroutine call. If Debug is unsuccessful up to this point, it then follows the control flow into the subroutine. It emulates all instructions up to the first conditional jump. By putting breakpoints at the destinations of this jump and allowing the program to execute until one of these breakpoints is reached, the flow of control is discovered. This process of emulating and setting breakpoints at conditional jumps is repeated until the flow returns to the original module or until Debug decides that it will be faster to get some help from the user. If the return address is still in the line being stepped over then Debug has a little more work to do before it finishes single-stepping the line. If Debug decides to give up, it finds the next line in the source text and asks the user if it is OK to proceed to this line. If the user knows that control will eventually transfer to this point then he may type a carriage return and Debug will proceed to this point. If the user knows that control flow will branch elsewhere, then he must set breakpoints and use the normal proceed command to get out of the unknown subroutine.

Emulation requires Debug to know about all the instructions the

machine can execute. For each instruction, a table describes where the next instruction will be executed and the number of conditional skips this instruction may perform. Debug also needs this information when it is requested to proceed from a location which has a permanent breakpoint. This is accomplished by replacing the permanent breakpoint with the original instruction and setting temporary breakpoints at all the destinations of this instruction. The user's program is then allowed to execute for this one instruction. When Debug regains control it puts back in the permanent breakpoint and removes the temporary ones. Now, if the program is not at another breakpoint, it is allowed to continue.

7. CONCLUSIONS

Debug/1000 has been demonstrated to be a very useful tool for program development. The productivity of the user goes up and frustration levels go down. Its feature set seems to be complete but not extravagant. Its computational requirements are small, thus little additional load is placed on the system. The users program usually runs at full speed until a breakpoint is reached. The system is required to have enough memory to allow the program being debugged to be locked into core.

One of the most useful features of Debug is its ability to display the source text while following the flow of a program. This enables the user to quickly understand what the program is doing. Once the program is understood, it is usually simple to correct the problem.

Program Profiling with Debug/1000

Debug/1000 provides a program profiling capability in addition to the features described in the previous article.

A program profiler is a tool used to identify what parts of your program are taking the most time, to help you make your program run faster. A symbolic debug profiler knows the subroutine names in the program. Therefore a histogram of which subroutines are taking the most time can be displayed.

The Debug profiling capability is a separate mode from the debug mode. You can initiate a debugging session and switch into the profiling mode.

Debug, while in the profiling mode, provides the following features:

- Plot a histogram of subroutines
- Plot a histogram of a subroutine
- Log profiling data to a specified file
- Display lines of source code
- Terminate profiling and exit Debug

In performing profiling functions, Debug alternately runs and waits at 10 msec intervals and samples the program location counter for your program at these intervals. Since it attributes the entire 10 msec time frame to your program, it is essential that no other programs take up this time. Therefore, you should use only a system that is not busy.

Following is a plot of a histogram of the subroutines of a sample program as output by Debug/1000.

```
Routine                 Amount   Histogram
-------                 ------   ---------
GIMME                     8%   ***************
ATOL                      7%   **************
GETCH                     5%   *********
WHERESLAVE                5%   *********
RPEEK                     4%   ********
IN                        4%   *******
ADROF                     4%   *******
CLEARB                    1%   **
MVW                       1%   **
LTOA                      1%   **
SETB                      1%   **
DOPOK                     1%   **
DEST                      1%   *
GTCLK                     1%   *
WINDX                     1%   *
SSTEP/DEBU1               1%   *
MBT                       1%   *
WHERE                     1%   *
Other (known code)        5%   *********
Other (unknown code)     18%   ******************************
I/O suspend              11%   ********************
Wait state               20%   ************************************
System noise              1%   *
```

In the sample histogram, the subroutines that spent the most amount of the total time are plotted with the busiest at the top. The percentage is rounded up to the nearest whole number which should be used only for comparison. Any of the subroutines that took up less than .5% of the execution time are not listed by name but are summed up and listed under "Other (known code)". These added up to 5% in the example plot. Time spent is subroutines which have been compiled with ASMB or FTN4 or have been modified by OLDRE appear in the "Other (unknown code)" line. Most of the system library routines fall into this category. The I/O suspend category includes time spent by the program waiting on unbuffered I/O devices, e.g., discs. The wait state is time spent while waiting on another program or on buffered I/O device suspend or other waiting conditions. These states are different for RTE-A.1 because there are different types and more wait states. System noise occurs whenever Debug samples the program and finds that it is pre-empted by the operating system.

To plot a histogram of subroutine ATOL in the sample program:

```
Overview? h atol
Profile for module ATOL

     7% of total time spent here

  Line No.       Amount   Histogram
  --------       ------   ---------
     764           2%   ****
     773           1%   *
     778           1%   *
     783           1%   **
     786           1%   **
     790           1%   *
     802           1%   **
     811           8%   ****************
     812          10%   ********************
     813           7%   ***************
     821           8%   ****************
     822          17%   ***********************************
     824           9%   ******************
     827          13%   ***************************
     840           1%   *
     843           1%   *
     848          18%   ************************************
     852           1%   *
     853           1%   **
     868           1%   *
```

In this plot, any line number that took up an execution time is listed, along with the corresponding per cent time. Comment lines and lines without any data samples are not listed.

HP AUTO ANSWERS

David Triggs, HP Systems Engineering, Sydney.

Many HP 1000 users have expressed interest in being able to dial into their computers from remote sites. This can be attractive in a number of situations. For example, suppose that the 1000 is located in a factory which is a long way from your office. You want to be able to use it without going out there each time. If you don't need to access the 1000 all the time the cost of a leased line probably can't be justified. However a dial in modem can provide a cost effective solution. Or as another example suppose that your work frequently takes you away from your office. With an accoustic coupler and a terminal you could take the capabilities of your computer with you. You can even work from home if you want. All that you need is a telephone.

Several 1000 users in Australia have already installed dial up modems and found them of great value. However others have had difficulty in understanding what they need to do to get a dial up modem going. Hopefully this article will clear up that confusion.

The first thing you need to do is select a modem and a method of connecting it to the 1000. As only Telecom supplied modems can be connected to the telephone system in Australia choosing a modem is just a matter of selecting which type of service (speed, etc.) is required. In order to talk to the 1000 the modem must be full duplex, asynchronous and of a baud rate supported by the 1000. While the 1000 will support many baud rates the only speed traditionally supported by Telecom was 300 baud. Fortunately now Telecom are also supporting 1200 baud. So either of these can be used. The application forms (called T76 forms) needed to obtain modem lines from Telecom should be obtained from Hewlett Packard. This is advised because you will need to include on these forms approval numbers etc. for the various pieces of equipment. If we know what you want to connect to what we can ensure that you have all the correct forms and information. The question of an interface is a little more complex. There are two basic contenders on M/E/F processors.

The first and most widely used is the 12966A BACI interface. This provides the modem signals required by most modems but is restricted to HP terminals. This is because the RTE driver for this card is only designed to work with terminals which use HP style ENQ/ACK handshaking. With this interface the modem is controlled by software in the 1000.

The other option is to use the 12792A Multiplexer (MUX). This device supports both HP and non-HP terminals, however it provides no modem control lines. If a modem or a modem combined with a black Box of some sort can be found that is sufficiently smart all control of the line can be done by the hardware. The 1000 can then treat the modem as a hard wired terminal and no additional software or programming is required in the 1000. It is worth noting that you can't just build your own black box. Any equipment connected to Telecom modems must have Telecom approval. At this time I am not aware of anyone in Australia having a Telecom approved installation of this type.

If you are using a BACI interface then the control of the modem lines must still be done by software. Fortunately the driver DVA05 will do most of the work for you. The driver provides a number of control requests to do things like setting the baud rate, automatically answering the phone when it rings and hanging up the phone. For more details see the DVR05/DVA05 reference manual. However they are not quite as simple to use as just putting a couple of CT requests in the WELCOM file, as a couple of users have found out. The best method is to write a simple program that just loops issuing a line open, waiting for a session to start and end and then issuing a line close to hang up the phone.

Some time ago I wrote a simple program to show several users what could be done. This was submitted by Corrado DiQual for the August issue of Crosstalk. Unfortunately I didn't see this copy before publication because there is a fairly obvious error in it which escaped detection. The "if" statement after statement 350 should contain an .ne. not an .eq. condition. This is a very simple minded program. Most of the code just logs messages to the console. I have subsequently added several bells and whistles such as finding the EQT and subchannel from the system tables, automatically configuring the commands for the MESSS calls and the capability to detect and correct several error conditions which the first version could not cope with. The techniques it uses should be easy to apply to meet a range of requirements. If anyone wishes a copy of the source for this program they should contact me as it is now too long to put in CROSSTALK. It is also starting to become too long to want to type in. The code is now written in FORTRAN 77 however with the addition of a few goto statements it could be modified to run with earlier fortran compilers: I couldn't bring myself to do it. The present version also assumes RTE-6 although it could also be ported to RTE-4B fairly easily.

The program makes use of a fair amount of information about the internals of the operating system. Some of this information is not part of the external interface of the operating system and as such is susceptible to change without notice. Essentially it just loops letting people log on and then off again watching for any error conditions. If any error state is detected and remains after a reasonable delay all I/O to the line is flushed and the line hung up. The logging on and off of users is detected with the IDGET call. This returns the ID segment number of the users FMGR or O if there is no ID segment, which will be the case before the user logs on and again after they log off. Error detection is achieved by using EXEC I/O so that error conditions can be handled from within the program and by watching the DRT and the EQT. There are two other techniques of note.

The first one is used to detect that a call has come in and that the driver is waiting for carrier. Because the line open request (32) does not terminate until carrier is received it is necessary to cheat. When a call is received the modem asserts the RING signal. At this point the driver starts polling to detect the arrival of carrier. This is done because the card can not interrupt on the arrival of carrier. The driver achieves polling by setting its own timeout clock to the polling interval and letting the operating system time it out. This can be detected by watching word 15 of the EQT.

The second one is used to flush pending requests to the modem line so that a hang up control request can be issued. This is a four step process. The first step is to down the EQT of the modem line. This forces all pending requests to be queued off the DRT instead of the EQT. The second step is to point the LU to the bit bucket with the SYLU command. All the requests are then performed on the bit bucket at a very high speed! Steps three and four involve restoring the LU and upping the EQT. This approach can be used for other devices as well. It is particularly handy for mag tapes.

Obviously this program is not going to do all the things that everyone wants. It should however provide a good start.

PROGRAM CLONING THE CIVILIZED WAY

Have you ever come across the need to clone and run programs in type 6 files and discovered that you must RP them first or use a FMGR transfer file?

Have you ever tried to run FMGR from FMGR in background to do something long and time consuming like packing 8000 track cartridges? You CAN'T do it, EH!

Here is a program (written in FORTRAN 77, but can be converted to other languages) which will clone and run a program from an existing ID segment or a type 6 file. Programs can be scheduled with wait or without wait (e.g. compiling large source files in background while you're doing something else).

Corrado DiQual,
(02) 699 0044

```
*{+-------------------------------------------------------
      program clone(),CDQ program cloner <821122,1342>
*********************************************************
*
      character*80 runstring,dummy
      integer*2 irunstring (40) ,rcpar ,comma
      equivalence (runstring,irunstring)
      integer*2 parameters (5)
*
      call rmpar(parameters)
* if no runstring go into interactive mode
      if( rcpar(1 ,runstring) .eq. 0) goto 10
* retrieve entire runstring
      if (rcpar(-1,runstring) .ne. 0 ) then
*        remove RU,CLONE, from string
         comma = index(runstring,',')
         comma = index(runstring(comma+1:),',')  + comma
         dummy = runstring(comma+1:)
         runstring =dummy
         goto 20
      endif
*
* interactive mode
 10   write(1,'("run what program?:_")')
      read(1,'(a80)') runstring
*
 20   call run_program(runstring,parameters)
      call prtn(parameters)
      write(1,'("new name=",a20)') runstring
*
      end
*
```

```
*{+-------------------------------------------------------+
*|Name : run_program                                      |
*|Type : subroutine                                       |
*|Language : fortran 77                                   |
*+-------------------------------------------------------+
*| ABSTRACT :                                             |
*|-----------                                             |
*|  This subroutine will clone a program or a type 6 file |
*|  giving it a new name that does not conflict with any  |
*|  system naming procedures.                             |
*|  Possible number of new names for each program = 8*26  |
*|  Subroutine will clean up id segments after program    |
*|  terminates  if the program was scheduled with wait .  |
*+-------------------------------------------------------+
*| CALLING SEQUENCE :                                     |
*|--------------------                                    |
*|    call run_program(runstring,parameters )             |
*|                     ---------  ----------             |
*|     where : runstring is of type character*   and      |
*|             contains the following :                   |
*|             progname,parm1,parm2,.........,etc         |
*|    fmgr namr's are valid as program names although only|
*|    security code and cartridge reference are used.     |
*|                                                        |
*| The runstring is passed to the son which can retrieve  |
*|     it using GETST library routine.                    |
*|                                                        |
*|     The name of the cloned program  is returned in     |
*|     runstring if cloning was succesfull.               |
*|                                                        |
*| If an error occurs,the error mnemonic is placed in the |
*| session control block.                                 |
*| This mnemonic can be retrieved by the calling          |
*| program using GTERR library routine or                 |
*| via the system HELP program.                           |
*|                                                        |
*| parameters is an integer array equivalent to fmgr 1p..5p|
*|       which may be retrieved by the son using RMPAR    |
*|     If the son returns these parameters using PRTN     |
*|      they will be returned                             |
*|      to the calling program in parameters array.       |
*|                                                        |
*| if the the first character of the runstring is a '&'   |
*|  then the program will be scheduled without wait.      |
*|  e.g. call run_program('ftn7x,&source,-,-')! with wait |
*|  e.g. call run_program ('&ftn7x,&source,-,-')! no wait |
*+-------------------------------------------------------+
*| PRE-REQUISITES :  none that I can think of             |
*|-------------                                           |
*+-------------------------------------------------------+
*| Create date :      12:22 PM  WED., 11  AUG., 1982      |
*| Programmer  :   Corrado Diqual                         |
*| Last change : <821122,1342>                            |
*| Revision date :                                        |
*| Revised  by :                                          |
*| Revision description :                                 |
*|                                                        |
*|   (this section to be repeated for every revision)     |
*+-------------------------------------------------------+}
*
*{+-------------------------------------------------------
      subroutine run_program(runstring,parm),CDQ <821122,1342>
     +  clone and run program and pass runstring parameters
*********************************************************
      implicit none
*
      character* (*) runstring
      character*6 new_name
      integer*2 parm(5),iscb_error(4),counter
      character*6 program_name
      character*8 scb_error
      character*80 drunstring
*
      integer*2 error,idcb(144),schedule_code,wait,nowait
      integer*2 name(10),security,cartridge,start_char
      integer*2 areg,breg,new_id
      integer*2 inew_name(3),iprogram_name(3)
      integer*2 irunstring(40) ,idget
*
* I've got to fiddle with integer arrays &characters
* because EXEC & FMP won't take character variables
      equivalence (name(5),security),(name(6),cartridge)
      equivalence (program_name,name,iprogram_name)
      equivalence (drunstring,irunstring)
      equivalence (new_name,inew_name)
      equivalence (scb_error,iscb_error)
*
      wait = 9+100000b
      nowait = 10+100000b
*
* if first character of runstring ='&' then program will be
* scheduled without wait
      call uppercase(runstring) ! convert to uppercase
      if (runstring(1:1) .eq. '&') then
          schedule_code = nowait
          drunstring = runstring(2:)!strip '&' from string
          runstring = drunstring
      else
           schedule_code =wait
      endif
*
      drunstring = runstring
*
      start_char = 1
      call namr(name,irunstring,80,start_char) !parse namr
      new_name(1:5) = program_name(1:5)
      new_name(4:6) = '#@ '
*
* Insert 'RU,' in the runstring because most programs
* which use the runstring expect FMGR to be the father.
*
          drunstring = 'RU,'//runstring
          runstring = drunstring
*
* First we'll check if an ID seg already exists
* with the name we're trying to use.
* This check is done because IDDUP & IDRPL do not return
* an error when you try to duplicate an existing ID
* segment,even though HP documentation says they will.
*
      call clone_new_name(new_name)
      do while  (idget(inew_name) .ne. 0)
          call clone_new_name(new_name)
      enddo
*
* If a cartridge reference is specified then
*  go to the specified file
      if (cartridge .ne. 0) goto 20
*
* if ID segment already exists try to clone the program
      call iddup(iprogram_name,inew_name,error)
*
      if (error .eq. 17 ) then ! program can't be copied
          new_name = program_name ! use original name
```

```
         goto 100 ! go and run the program
      endif
*
      if (error .eq. 0) goto 100 ! go and run the program
*
* if we get here then the program is probably in a type 6
* file so we'll try to open the file non-exclusively
 20   call open(idcb,error,iprogram_name,1,security
     + ,cartridge)
      if (error .eq.-6) error = 67 ! program not found
      if (error .ne. 6) goto 9000 ! fmp errors
*
* now try and restore the program
      call idrpl(idcb,error,inew_name,new_id)
*     if (error .ne. 0) goto 9000
*
* close the file
      call close(idcb,error)
      if (error .ne. 0) goto 9000
*
*
* now we'll try and run the program
 100  call exec(schedule_code,inew_name,parm,
     + irunstring,-len(runstring),*9002)
*
* Retrieve any parameters passed back by the son and
* return them to the calling program.
      call rmpar(parm)
*
*
* if program was scheduled with wait we'll get
*  rid of the id segment
*
      if (schedule_code .eq. wait) then
          call idrpd(inew_name,error)
          if (error .ne. 0 ) goto 9000
      endif
* if program was scheduled with no wait
* let the caller manage the id segment
* so put the name of the cloned program into runstring
      runstring = new_name
      return
*
* error reporting routines
 9000 write (scb_error,'("FMGR",i4,3)') error
      write (1,'("FMGR",i4,3)') error
* Place the error mnemonic in the session control block
      call pterr(iscb_error)
      return
*
 9002 call abreg(areg,breg)
      write (scb_error,'(2a2)') areg,breg
      write (1,'("EXEC error=",a2,a2)') areg,breg
* Place the error mnemonic in the session control block
      call pterr(iscb_error)
      return
*
      end
*
*(+*************************************************************
      subroutine clone_new_name(new_name),CDQ <821122.1342>
*************************************************************
*
      implicit none
      character*6 new_name
      integer*2 counter ,char4,char5
*
* replace all spaces in string with '#'
      do counter = 1,3
          if (new_name(counter:counter) .eq. ' ') then
              new_name(counter:counter) ='#'
          endif
      enddo
      char5 = ichar(new_name(5:5)) +1
      if ((char5 .gt. 90) .or. (char5 .lt. 65 )) then
          char5 = 65
          char4 =ichar(new_name(4:4)) + 1
          if ((char4 .lt. 35) .or. (char4 .gt.42 )) then
              char4 = 35
          endif
          new_name(4:4) =char(char4)
      endif
      new_name(5:5) = char(char5)
      end
*
*(+*************************************************************
      subroutine uppercase (string),CDQ <821122.1342>
*************************************************************
*
      character* (*) string
      integer*2 count
*
      do count = 1, len(string)
          if ((string(count:count) .ge. 'a') .and.
     +        (string(count:count) .le. 'z')) then
              string(count:count)
     +          = char(ichar(string(count:count))-32)
          endif
      enddo
      end
```

A comparison between the User Interfaces of the HP 1000 and the VAX 11/750

by Jeffrey Deakin,
Australian Coal Industry Research Laboratories Ltd.

1. INTRODUCTION

The purpose of this article is to present a subjective comparison between the ways in which the HP 1000 and VAX-11 family present themselves to the end user. While particular reference is made to the ACIRL VAX 11/750, what is said applies to the other members of the VAX family, which share a common operating system, VMS.

Over the years, I've used a number of different systems, each containing features not found in the others. Comparisons are very difficult; it is hard to do each system justice, and they can really only be made in the context in which the different systems were developed. For example, a common criticism of the HP 1000 is that the user talks to the FMGR program, not RTE directly, and that RTE doesn't know about files. But this seemingly odd arrangement is reasonable when one considers what RTE evolved from, and what its original purpose was. I personally think that if the HP team had the advantage of starting from scratch, they would come up with something like a UNIX or VMS look-alike, for these are the operating systems of the 80's. In thinking ahead, though, HP should look to the 90's as well, when systems along the lines of the Smalltalk environment might become more common.

2. THE ACIRL APPLICATION

Before commencing the comparison, I will present a brief description of the ACIRL system and application.

The hardware includes:
— a 1 Mbyte 11/750 computer
— 240 Mbytes of disc (2 × RM80)
— dual density (800/1600 bpi) 125 ips TU77 magnetic tape unit
— 5 VT100 screen terminals, including one with graphics
— 2 hardcopy Decwriter terminals (one is the operator console)
— 1 Tektronix 4114 high resolution (1020 by 760) graphics terminal, which also acts as the system interface with:
  — a Tektronix 4662 8 pen plotter
  — a Calcomp 960 digitizing table
— 1 Calcomp 1051 drum plotter
— a remote HP9826 desktop computer situated at ACIRL's Bellambi laboratory, connected by a 1200 bps leased line. It is used for simultaneously controlling and logging 3 areas of testing, namely rock testing, physical strata model, and the creep testing rig.

As the nature of the hardware suggests, the software developed by the ACIRL Mining Division makes extensive use of graphics output. Our software systems, which are written in Fortran 77, encompass the following areas:
— a geotechnical logging system
— more ventilation simulation
— structural stability programs
— in-seam long hole drilling program
— underground mine plan generating system
— washability performance prediction.

3. COMPARISON BETWEEN USER INTERFACES

From the user's point of view, a system's features manifest themselves in the areas of routine use, and program development. With respect to the former, RTE and VMS are not all that different, in the sense that:
— execution speed is much the same
— program invocation is similar.

For example, to produce a particular picture on our VAX, one types:
$ mpdraw/scale=8000/er=216000,217000/nr=1307500, 1308500 acme

The corresponding HP 1000 command might look like:
:mpdrw,acme,scale=8000,er=216000,217000,nr=1307500,
1308500

From the programmer's point of view, the development environment is important because it can profoundly effect the rate at which we progress through the edit-compile-link cycle, which is the characteristic limitation of the current generation of software. Because the development user is the one most likely to notice system differences, and also because Fortran is the language we use, I will restrict my attention to the Fortran programming environment.

The systems side of the machines are not considered, although it is worth mentioning in passing that the VAX provides some very convenient facilities for setting up your own queues with forms types, etc., and backup utilities that allow only files that have been changed since the last backup to be saved.

3.1 THE SYSTEM ENVIRONMENT

Essentially, productive program development requires an Editor, a compiler, a linking loader, a debugger and a friendly system environment. This section deals with the latter feature, in terms of a number of specific topics.

HELP

Both systems provide on-line help, however the VAX help is more extensive and has the same format right across the system, consisting of a tree whose branches you traverse as you require more detail. The extent to which help is provided gives a very friendly impression; significant portions of the manual are stored in the system help library.

GENERAL

VMS tends to be excessively verbose if anything, while RTE is more cryptic. For example, when a program crashes, RTE gives a brief message informing the user that a DM or MP error has occurred, the contents of various registers, and the address at which the error occurred. The user would then go back to the loader and computer listings, and locate the offending code. The VAX provides a lengthy verbal diagnostic and (what is good) a traceback sequence displaying the current line number for all modules back to the main program. Together with the contents of registers the assembly programmer is really only interested in seeing, you get about 6 lines in all. What makes it painful sometimes is that lengthy error messages occur even if minor formatting errors occur. While the purist would argue that this is a sign of system integrity, it is often a bit much. The HP formatter is more forgiving and flexible, and even lets you write reals with "I" format.

The VAX allows you to define symbols, which are useful as typing savers and also lets you define your own commands (cf macros in the SH program). The user can also define things called logical names which are very useful in command procedures. Both systems allow the user to pass parameters to command procedures.

TERMINAL INTERFACE

VMS allows typeahead, while you can only do it on the HP if it is equipped with a MUX. FMGR allows the user to edit the pending line, a very handy feature not found in VMS.

If a long listing is flashing past on the screen at 9600 baud and you want to read part of it, the VAX allows control characters to alternately stop and start output (CTRL S(XON) and CTRL Q (XOFF)), to suppress output (CTRL O), or to abort the listing/process (CTRL Y). CTRL Y can be trapped in a command procedure, for example:

```
$ on control-y then goto cleanup
.
.
.
$ exit
$ cleanup:write sys$output "control Y intercepted"
.
.
.
```

Unfortunately, HP have written a lot of these control characters into their editor, which means that standard characters like XON and XOFF can't be used for their more common use, handshaking. The only way to suppress output is to bash the keyboard until the "S=nn COMMAND?" prompt appears. While this seems to lack elegance, I used this to my advantage when a regular HP user, to do several things like edit, compile and link different routines simultaneously.

FILES

The VMS file system is quite different to that of FMGR. On the HP, each user has a specific fixed area of contiguous disc tracks available to him or her alone, called a cartridge. The cartridge has an associated logical unit number and cartridge reference number (2 ASCII characters), and a file directory located in its last track or tracks. In the system cartridge, a central cartridge directory points to the user cartridges. On the VAX, each user is allotted a quota of disc blocks on each disc device; these blocks can exist anywhere. (A disadvantage of this arrangement is that it's harder to undelete deleted files. An advantage is that it's more efficient). The user is assigned a directory which can be partitioned into subdirectories and sub-subdirectories to any level, corresponding to the user's hierarchy of activities. File names within the directory are of the form NAME.TYPE;VERSION. NAME is a 1-9 character name, TYPE is a 3 character file type, and VERSION is a number, the highest corresponding to the "latest". Unlike FMGR, non-standard characters like * or/ aren't permitted in the file name, but having 9 characters plus a type compensates for this deficiency. The file type is used to designate a file's purpose: for example, .DAT, .FOR, .OBJ, and .EXE are used for data, Fortran source, object code and executable code respectively. Thus if you say $ fortran prog or $ run prog the system looks for PROG.FOR or PROG.EXE respectively. We have a few standard file types of our own, for example .PLT for neutral plot files.

Having version numbers is handy because when you edit a file you get to keep the previous version as well as the new version. In this context, the Fortran STATUS='NEW' and 'OLD' have very well defined meanings. The number of versions the user has is a system parameter; older versions are automatically deleted. The "purge" command gets rid of all but the latest version.

Like FMGR, VMS lets you do "wild card" directory lists, but you can also do wild card deletes, purges, and copies, etc.

The VAX supports 3 file organizations: sequential, relative (fixed length record, direct access), and indexed (keyed access). FMGR supports only the first two of these: the third is something you only get with other languages and is very handy if you're using complex data records. Keys can be primary or alternate, character or integer.

3.2 PROGRAM DEVELOPMENT

GETTING THE PROGRAM IN — THE EDITOR

The two editors, EDIT/1000 and EDT share the features of help, line mode and screen mode. In line mode, they're much alike, both having their roots somewhere in UNIX. EDIT/1000 supports regular expressions, while EDT doesn't, although you can use the other VMS editor "SOS", if you need that capability. Both allow you to recover from interrupted sessions (system crashes) by writing a journal file. EDIT/1000 lets you "undo" the previous command, which is rather a nice feature. To do this on the VAX you have to interrupt the Editor (using CTRL Y), edit the journal file and then EDIT/RECOVER.

My productivity has significantly increased since I began to use VMS, and I think it is due to EDT's screen mode. In screen mode, the two editors are quite different, reflecting their different design philosophies. EDT uses the CPU and a "fairly dumb" terminal (the VT100), while EDIT/1000 uses block reads and the local editing capabilities of the "fairly smart" HP26xx terminals. EDT is more expensive to run but gives you capabilities you just can't have unless your terminal happens to be a microcomputer with its own screen editor. EDT screen features include:

— cut and paste (character/word/line/section)
— move to next character/word/line/section, forwards or backwards
— delete/undelete character/word/line
— buffers for storing text
— search for strings, forwards or backwards
— automatic search for/substitute strings

COMPILING THE PROGRAM — FORTRAN

Both systems support superb Fortran 77 compilers. Now that HP support the character data type, they're no longer behind the others. (After all, it is 1982!). About the only difference I have noticed is that VAX Fortran gives you a cross-reference listing if you want it: for every entry in the symbol table, a list is produced containing all line numbers in the program where that entry appeared, and whether it was a data or assignment statement, or subroutine call. Of course, VAX Fortran is verbose: e.g., the keyword in the OPEN statement to suppress Fortran carriage control

attributes in a new file is CARRIAGECONTROL='LIST', a mouthful that should be abbreviated.

LINKING THE PROGRAM

Both systems have linking loaders with similar features. The VMS linker appears to be faster than the RTE-IVB loader, although I'm told that the RTE-6 loader is a lot faster. Both allow libraries to be searched for unresolved references. The VMS linker has one deficiency here. Suppose MAIN calls SUB1 and SUB2, and SUB2 calls FRED, and that SUB1 and FRED are located in LIB1, and SUB2 is located in LIB2. Then the command

$ link main,lib1/lib,lib2/lib

will fail to resolve the reference FRED because the linker does not look at previously searched libraries. I think the HP 1000 does automatically do this. To get around the problem, we have to use the command $ link main,lib1/lib/include=fred,lib2/lib

One good application of logical names (mentioned earlier) is that there are up to 1000 system logical names called LNK$LIBRARY xxx. If such a logical name is defined to be some central storeable library of commonly used utilities; then the linker automatically searches that library if there are any unresolved references. Thus all you have to type is $ link main.

DEBUGGING THE PROGRAM

Since I have never really used the RTE Debugger, I can't really comment on it except to say that it is not a symbolic debugger (although it soon will be!), and appeared to be unfriendly (and hence discouraging)! However, the VAX debugger DBG is very nice, and I think it is worth a mention. Again you get the full help facility, ability to refer to program symbols, line numbers and statement labels. You can even examine the source code! The examples below indicate the sort of stuff DBG is made of:

deposit kount=5	examine/hex char
examine x(1):x(n),z	cancel breaks/all
help set module	show breaks
set break % label 80	show modules
examine/source %line 551	go
cancel break %line 551	

At ACIRL we find that the debugger speeds up program development quite significantly, and people tend to use it because of its general friendliness and ease of use.

4. CONCLUDING REMARKS

Despite a number of shortcomings, VMS is very nice to use, and has done much to shorten the program development process. The Systems Improvement Committee would do well to examine VMS, along with other systems, in the course of their study.

HP 1000 Users Group (N.S.W.)

The Annual General Meeting of the Group was held on Wednesday, December 1 at Hewlett Packard, North Ryde. Elected Office bearers for 1983 are:

Corrado Di Qual, President, STC
Dr. N. T. Hai, Treasurer, Lever & Kitchen
Bill Wallace, Secretary, O.S.S.
Bill Filson, Committee, Dasys
Ralph Davies, Committee, Plessey
Tony Lane, Committee, Cullen, Egan Dell

Prior to the election, John Knaggs (HP) presented an entertaining talk on the new 32 bit HP9000, with particular reference to its innovative hardware features. General business, after the election, included discussion of possible workshops organised by the Users on aspects of HP Software, etc. A very pleasant evening was topped off by an informal Christmas party by courtesy of Hewlett Packard.

The N.S.W. Group wishes Crosstalk subscribers the compliments of the season.

Bill Wallace,
(Secretary)

SPECIFICATIONS FOR SUBMISSION OF ARTICLES AND ADVERTISEMENTS

Crosstalk will be published each even numbered month. Articles and advertisements must be received by the appropriate group editor by the third week of the preceding month.

ARTICLES: Articles should be typed with any diagrams and program listings in camera-ready form (Author's name, address and phone number should be included).

ADVERTISEMENTS: Display ads. should be in camera-ready artwork form. The printer may be instructed to layout ordinary typeface ads.

CURRENT ADVERTISING RATES:
Full page — $250
Half page — $125
Column/cm — $4

There is a 20% discount on these rates for regular advertisers. Classified ads. are free for user group members, and $10 each for non-members.

Advertisers will be billed upon receipt of ad. The user groups reserve the right to change rates, limit space availability and reject advertising which is deemed inappropriate.

ADDRESSES FOR SUBMISSION OF ARTICLES AND ADVERTISEMENTS:

The Editor,
HP1000 Users Group (N.S.W.),
Box 3060 GPO,
Sydney, 2001.
N.S.W.

The Editor,
HP1000 Users Group (Vic.)
P.O. Box 132,
Mt. Waverley, 3149
Vic.

Ms. Barbara Harrison,
Canberra Technical Users Group
C/- Australian National Parks & Wildlife,
4th Floor, Adelaide House,
Phillip, A.C.T., 2606.
Phone: (062) 897 919.

Mr Keith Crellin,
Queensland Technical Users Group,
C/- Cameron McNamara Pty. Ltd.,
131 Leichhardt Street,
Spring Hill, Qld., 4001.
Phone: (07) 228 9125.

H.P.D.C.U.G.V. articles only to:
Mr Bernie O'Shannessy,
Arlec,
30-32 Lexton Road,
Box Hill, 3128, Vic.

H.P.D.C.U.G.V. advertisements only to:
Advertising Editor,
HP Desktop Computer Users Group (Vic.),
C/- 47 Bursaria Ave.,
Ferntree Gully, 3156, Vic.

COMING EVENTS

Jan. 24:	IMAGE/DBMS 1000 Course, HP Melbourne
Feb. 7:	RTE-A System User Course, HP Sydney
Feb. 14:	RTE-A System Programmer/Designer Course, HP Sydney
Feb. 14:	HP Basic Programming Course, HP Melbourne
Feb. 21:	9845 Operating & Programming Course, HP Sydney
Feb. 21:	9826/36 (Series 200) Operating & Programming Course, HP Melbourne
Feb. 28:	RTE-6/VM Session Monitor Course, HP Sydney
March 1:	H.P.D.C.U.G.V. Meeting — "Strings and Things", 4 pm., C.S.I.R.O., Glen Waverley
March 14:	9826/36 (Series 200) Operating & Programming Course, HP Sydney
March 21:	RTE-6/VM System Manager Course, HP Sydney
March 21:	PASCAL Course, HP Melbourne
March 21:	HP85 Basic Programming Course, HP Melbourne

CLASSIFIED ADVERTISEMENTS

FOR SALE OR HIRE

Computers & (ROMs) —

9845A 64k (IO,MM)	**$6000**
9831A 16k (MM)	**$2500**
9831A 32k	**$2500**
9825A 24k (SA, PG, SP)	**$3800**
9825T 64k	**$6800**

Printers —

9871A, attach, 25-45I/F	**$1400**
9871A alone (parallel)	**$750**
2631A alone (HP IB)	**$2200**

Other peripherals —

9885M & I/F (Master)	**$3300**
9885S (Slave)	**$2200**
9869A Card Reader	**$500**

Used, good disketts and tapes.

Units or Systems for HIRE.

Other equipment is available from clients registered in my used equipment FORUM.

CALL: CHRIS SIMPSON
(03) 859 6643

WANTED:

Surplus HP desktop computers, peripherals, INTERFACES, ROMs, etc.

Call: Chris Simpson

(03) 859 6643

CANBERRA TECHNICAL USERS GROUP NEWS

The last meeting was held on 28 September at Data Science's office. John Rees described and demonstrated GEODS/1000 — Exploration and Mining Geology Data-Base and Drafting System, and other software packages for the HP1000.

Grant Spratt and Greg Atkinson from HP described HP's latest offerings in the technical computer and instrument lines respectively. These included 30 channel D.A.S., HPIL, D.V.M., 9836 software enhancements, 2700 Series colour terminals, HP75C micro.