HEWLETT-PACKARD
JOURNAL

JANUARY 1986

HEWLETT-PACKARD JOURNAL

January 1986 Volume 37 • Number 1

Articles

4 **Compilers for the New Generation of Hewlett-Packard Computers,** *by Deborah S. Coutant, Carol L. Hammond, and Jon W. Kelley* *Optimizing compilers realize the potential of the new reduced-complexity architecture.*

6 **Components of the Optimizer**
16 **An Optimization Example**

18 **Authors**

20 **A Stand-Alone Measurement Plotting System,** *by Thomas H. Daniels and John Fenoglio* *Measure and record low-frequency phenomena with this instrument. It also can send the measurements to a host computer and plot data taken by other instruments.*

22 **Eliminating Potentiometers**

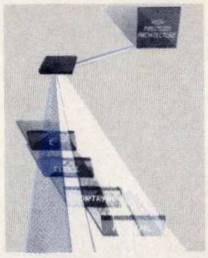
24 **Digital Control of Measurement Graphics,** *by Steven T. Van Voorhis* *Putting a microprocessor in the servo loop is a key feature. A vector profiling algorithm is another.*

27 **Measurement Graphics Software,** *by Francis E. Bockman and Emil Maghakian* *This package simplifies measuring, recording, plotting, and annotating low-frequency phenomena.*

32 **Analog Channel for a Low-Frequency Waveform Recorder,** *by Jorge Sanchez* *No potentiometers are used in this design that automatically zeros and calibrates its input ranges.*

36 **Usability Testing: A Valuable Tool for PC Design,** *by Daniel B. Harrington* *Sometimes a personal computer feature isn't used the way it was expected. Watching sample users can help remedy such difficulties.*

In this Issue



Hewlett-Packard's next-generation computers are now under development in the program code-named Spectrum, and are scheduled to be introduced in 1986. In our August 1985 issue, Joel Birnbaum and Bill Worley discussed the philosophy and the aims of the new computers and HP's architecture, which has been variously described as reduced-complexity, reduced instruction set computer (RISC), or high-precision. Besides providing higher performance than existing HP computers, an important objective for the new architecture is to support efficient high-level language development of systems and applications software. Compatibility with existing software is another important objective. The design of high-level language compilers is extremely important to the new computers, and in fact, the architecture was developed jointly by both hardware and software engineers. In the article on page 4, three HP compiler designers describe the new compiler system. At introduction, there will be Fortran, Pascal, COBOL, and C compilers, with others to become available later. An optional component of the compiler system called the optimizer tailors the object code to realize the full potential of the architectural features and make programs run faster on the new machines. As much as possible, the compiler system is designed to remain unchanged for different operating systems, an invaluable characteristic for application program development. In the article, the authors debunk several myths about RISCs, showing that RISCs don't need an architected procedure call, don't cause significant code expansion because of the simpler instructions, can readily perform integer multiplication, and can indeed support commercial languages such as COBOL. They also describe millicode, HP's implementation of complex functions using the simple instructions packaged into subroutines. Millicode acts like microcode in more traditional designs, but is common to all machines of the family rather than specific to each.

The article on page 20 introduces the HP 7090A Measurement Plotting System and the articles on pages 24, 27, and 32 expand upon various aspects of its design. The HP 7090A is an X-Y recorder, a digital plotter, a low-frequency waveform recorder, and a data acquisition system all in one package. Although all of these instruments have been available separately before, for some measurement applications where graphics output is desired there are advantages to having them all together. The analog-to-digital converter and memory of the waveform recorder extend the bandwidth of the X-Y recorder well beyond the limits of the mechanism (3 kHz instead of a few hertz). The signal conditioning and A-to-D conversion processes are described in the article on page 32. The servo design (page 24) is multipurpose—the HP 7090A can take analog inputs directly or can plot vectors received as digital data. A special measurement graphics software package (page 27) is designed to help scientists and engineers extend the stand-alone HP 7090A's capabilities without having to write their own software.

No matter how good you think your design is, it will confound some users and cause them to circumvent your best efforts to make it friendly. Knowing this, HP's Personal Computer Division has been conducting usability tests of new PC designs. Volunteers who resemble the expected users are given a series of tasks to perform. The session is videotaped and the product's designers are invited to observe. The article on page 36 reports on the sometimes humorous and always valuable results.

-R.P. Dolan

What's Ahead

The February issue will present the design stories of three new HP instrument offerings. The cover subject will be the HP 5350A, HP 5351A, and HP 5352A Microwave Frequency Counters, which use gallium arsenide hybrid technology to measure frequencies up to 40 GHz. Also featured will be the HP 8757A Scalar Network Analyzer, a transmission and reflection measurement system for the microwave engineer, and the HP 3457A Multimeter, a seven-function, 3½-to-6½-digit systems digital voltmeter.

The HP Journal encourages technical discussion of the topics presented in recent articles and will publish letters expected to be of interest to our readers. Letters must be brief and are subject to editing. Letters should be addressed to: Editor, Hewlett-Packard Journal, 3000 Hanover Street, Palo Alto, CA 94304, U.S.A.

Compilers for the New Generation of Hewlett-Packard Computers

Compilers are particularly important for the reduced-complexity, high-precision architecture of the new machines. They make it possible to realize the full potential of the new architecture.

by Deborah S. Coutant, Carol L. Hammond, and Jon W. Kelley

WITH THE ADVENT of any new architecture, compilers must be developed to provide high-level language interfaces to the new machine. Compilers are particularly important to the reduced-complexity, high-precision architecture currently being developed at Hewlett-Packard in the program that has been code-named Spectrum. The Spectrum program is implementing an architecture that is similar in philosophy to the class of architectures called RISCs (reduced instruction set computers).[1] The importance of compilers to the Spectrum program was recognized at its inception. From the early stages of the new architecture's development, software design engineers were involved in its specification.

The design process began with a set of objectives for the new architecture.[2] These included the following:

- It must support high-level language development of systems and applications software.
- It must be scalable across technologies and implementations.
- It must provide compatibility with previous systems.

These objectives were addressed with an architectural design that goes beyond RISC. The new architecture has the following features:

- There are many simple instructions, each of which executes in a single cycle.
- There are 32 high-speed general-purpose registers.
- There are separate data and instruction caches, which are exposed and can be managed explicitly by the operating system kernel.
- The pipeline has been made visible to allow the software to use cycles normally lost following branch and load instructions.
- Performance can be tuned to specific applications by adding specialized processors that interface with the central processor at the general-register, cache, or main memory levels.

The compiling system developed for this high-precision architecture* enables high-level language programs to use these features. This paper describes the compiling system design and shows how it addresses the specific requirements of the new architecture. First, the impact of high-level language issues on the early architectural design decisions is described. Next, the low-level structure of the compiling system is explained, with particular emphasis on areas that have received special attention for this architecture: program analysis, code generation, and optimization. The paper closes with a discussion of RISC-related issues and how they have been addressed in this compiling system.

*The term "high-precision architecture" is used because the instruction set for the new architecture was chosen on the basis of execution frequency as determined by extensive measurements across a variety of workloads.

Designing an Architecture for High-Level Languages

The design of the new architecture was undertaken by a team made up of design engineers specializing in hardware, computer architecture, operating systems, performance analysis, and compilers. It began with studies of computational behavior, leading to an initial design that provided efficient execution of frequently used instructions, and addressed the trade-offs involved in achieving additional functionality. The architectural design was scrutinized by software engineers as it was being developed, and their feedback helped to ensure that compilers and operating systems would be able to make effective use of the proposed features.

A primary objective in specifying the instruction set was to achieve a uniform execution time for all instructions. All instructions other than loads and branches were to be realizable in a single cycle. No instruction would be included that required a significantly longer cycle or significant additional hardware complexity. Restricting all instructions by these constraints simplifies the control of execution. In conventional microcoded architectures, many instructions pay an overhead because of the complexity of control required to execute the microcode. In reduced-complexity computers, no instruction pays a penalty for a more complicated operation. Functionality that is not available in a single-cycle instruction is achieved through multiple-instruction sequences or, optionally, with an additional processor.

As the hardware designers began their work on an early implementation of the new architecture, they were able to discover which instructions were costly to implement, required additional complexity not required by other instructions, or required long execution paths, which would increase the cycle time of the machine. These instructions were either removed, if the need for them was not great, or replaced with simpler instructions that provided the needed functionality. As the hardware engineers provided feedback about which instructions were too costly to in-

clude, the software engineers investigated alternate ways of achieving the same functionality.

For example, a proposed instruction that provided hardware support for a 2-bit Booth multiplication was not included because the additional performance it provided was not justified by its cost. Architecture and compiler engineers worked together to propose an alternative to this instruction. Similarly, several instructions that could be used directly to generate Boolean conditions were deleted when they were discovered to require a significantly longer cycle time. The same functionality was available with a more general two-instruction sequence, enabling all other operations to be executed faster.

The philosophy of reduced-complexity computers includes the notion that the frequent operations should be fast, possibly at the expense of less frequent operations. However, the cost of an infrequent operation should not be so great as to counterbalance the efficient execution of the simple operations. Each proposed change to the architectural specification was analyzed by the entire group to assess its impact on both software and hardware implementations. Hardware engineers analyzed the instruction set to ensure that no single instruction or set of instructions was causing performance and/or cost penalties for the entire architecture, and software engineers worked to ensure that all required functionality would be provided within performance goals. Compiler writers helped to define conditions for arithmetic, logical, and extract/deposit instructions, and to specify where carry/borrow bits would be used in arithmetic instructions.

As an example of such interaction, compiler writers helped to tune a conditional branch nullification scheme to provide for the most efficient execution of the most common branches. Branches are implemented such that an instruction immediately following the branch can be executed before the branch takes effect.[1] This allows the program to avoid losing a cycle if useful work is possible at that point. For conditional branches, the compiler may or may not be able to schedule an instruction in this slot that can be executed in both the taken-branch and non-taken-branch cases. For these branches, a nullification scheme was devised which allows an instruction to be executed only in the case of a taken branch for backward branches, and only in the case of a non-taken branch for forward branches. This scheme was chosen to enable all available cycles to be used in the most common cases. Backward conditional branches are most often used in a loop, and such branches will most often be taken, branching backwards a number of times before falling through at the end of the iteration. Thus, a nullification scheme that allows this extra cycle to be used in the taken-branch case causes this cycle to be used most often. Conversely, for forward branches, the nullification scheme was tuned to the non-taken-branch case. Fig. 1 shows the code generated for a simple code sequence, illustrating the conditional branch nullification scheme.

Very early in the development of the architectural specification, work was begun on a simulator for the new computer architecture and a prototype C compiler. Before the design was frozen, feedback was available about the ease with which high-level language constructs could be translated to the new instruction set. The early existence of a prototype compiler and simulator allowed operating system designers to begin their development early, and enabled them to provide better early feedback about their needs, from the architecture as well as the compiler.

```
(a)
L1  LDW              4(sp),r1       ; First instruction of loop
    :
    :
    COMIBT,>=,N      10,r2,L1+4     ; Branch to L1+4 if 10 >= r2
    LDW              4(sp),r1       ; Copy of first loop instruction,
                                    ; executed before branch takes effect

(b)
    COMIBF,=,N       0,r1,L1        ; Branch if r1 is not equal to 0
    ADDI             4,r2,r2        ; First instruction of then clause
    :
    :
L1  :
    :
```

Fig. 1. *An illustration of the conditional branch nullification scheme. (a) The conditional branch at the end of a loop will often be followed by a copy of the first instruction of the loop. This instruction will only be executed if the branch is taken. (b) The forward conditional branch implementing an if statement will often be followed by the first instruction of the then clause, allowing use of this cycle without rearrangement of code. This instruction will only be executed if the branch is not taken.*

At the same time, work was begun on optimization techniques for the new architecture. Segments of compiled code were hand-analyzed to uncover opportunities for optimization. These hand-optimized programs were used as a guideline for implementation and to provide a performance goal. Soon after the first prototype compiler was developed, a prototype register allocator and instruction scheduler were also implemented, providing valuable data for the optimizer and compiler designers.

Compiling to a Reduced Instruction Set

Compiling for a reduced-complexity computer is simplified in some aspects. With a limited set of instructions from which to choose, code generation can be straightforward. However, optimization is necessary to realize the full advantage of the architectural features. The new HP compiling system is designed to allow multiple languages to be implemented with language-specific compiler front ends. An optimization phase, common to all of the languages, provides efficient register use and pipeline scheduling, and eliminates unnecessary computations. With the elimination of complex instructions found in many architectures, the responsibility for generating the proper sequence of instructions for high-level language constructs falls to the compiler. Using the primitive instructions, the compiler can construct precisely the sequence required for the application.

For this class of computer, the software architecture plays a strong role in the performance of compiled code. There is no procedure call instruction, so the procedure calling sequence is tuned to handle simple cases, such as *leaf routines* (procedures that do not call any other procedures), without fixed expense, while still allowing the complexities of nested and recursive procedures. The saving of registers at procedure call and procedure entry is depen-

(continued on page 7)

Components of the Optimizer

The optimizer is composed of two types of components, those that perform data flow and control flow analysis, and those that perform optimizations. The information provided by the analysis components is shared by the optimization components, and is used to determine when instructions can be deleted, moved, rearranged, or modified.

For each procedure, the control flow analysis identifies basic blocks (sequences of code that have no internal branching). These are combined into intervals, which form a hierarchy of control structures. Basic blocks are at the bottom of this hierarchy, and entire procedures are at the top. Loops and if-then constructs are examples of the intermediate structures.

Data flow information is collected for each interval. It is expressed in terms of resource numbers and sequence numbers. Each register, memory location, and intermediate expression has a unique resource number, and each use or definition of a resource has a unique sequence number. Three types of data flow information are calculated:

- Reaching definitions: for each resource, the set of definitions that could reach the top of the interval by some path.
- Exposed uses: for each resource, the set of uses that could be reached by a definition at the bottom of the interval.
- UNDEF set: the set of resources that are not available at the top of the interval. A resource is available if it is defined along all paths reaching the interval, and none of its operands are later redefined along that path.

From this information, a fourth data structure is built:

- Web: a set of sequence numbers having the property that for each use in the set, all definitions that might reach it are also in the set. Likewise, for each definition in the set, all uses it might reach are also in the set. For each resource there may be one or many webs.

Loop Optimizations

Frequently the majority of execution time in a program is spent executing instructions contained in loops. Consequently, loop-based optimizations can potentially improve execution time significantly. The following discussion describes components that perform loop optimizations.

Loop Invariant Code Motion. Computations within a loop that yield the same result for every iteration are called loop invariant computations. These computations can potentially be moved outside the loop, where they are executed less frequently.

An instruction inside the loop is invariant if it meets either of two conditions: either the reaching definitions for all its operands are outside the loop, or its operands are defined by instructions that have already themselves been identified as loop invariant. In addition, there must not be a conflicting definition of the instruction's target inside the loop. If the instruction is executed conditionally inside the loop, it can be moved out only if there are no exposed uses of the target at the loop exit.

An example is a computation involving variables that are not modified in the loop. Another is the computation of an array's base address.

Strength Reduction and Induction Variables. Strength reduction replaces multiplication operations inside a loop with iterative addition operations. Since there is no hardware instruction for integer multiplication in the architecture, converting sequences of shifts and adds to a single instruction is a performance improvement. Induction variables are variables that are defined inside the loop in terms of a simple function of the loop counter.

Once the induction variables have been determined, those that are appropriate for this optimization are selected. Any multiplications involved in the computation of these induction variables are replaced with a COPY from a temporary. This temporary holds the initial value of the function, and is initialized preceding the loop. It is updated at the point of all the reaching definitions of the induction variable with an appropriate addition instruction. Finally, the induction variable itself is eliminated if possible.

This optimization is frequently applied to the computation of array indices inside a loop, when the index is a function of the loop counter.

Common Subexpression Elimination

Common subexpression elimination is the removal of redundant computations and the reuse of the one result. A redundant computation can be deleted when its target is not in the UNDEF set for the basic block it is contained in, and all the reaching definitions of the target are the same instruction. Since the optimizer runs at the machine level, redundant loads of the same variable in addition to redundant arithmetic computations can be removed.

Store-Copy Optimization

It is possible to promote certain memory resources to registers for the scope of their definitions and uses. Only resources that satisfy aliasing restrictions can be transformed this way. If the transformation can be performed, stores are converted to copies and the loads are eliminated. This optimization is very useful for a machine that has a large number of registers, since it maximizes the use of registers and minimizes the use of memory.

For each memory resource there may be multiple webs. Each memory web is an independent candidate for promotion to a register.

Unused Definition Elimination

Definitions of memory and register resources that are never used are removed. These definitions are identified during the building of webs.

Local Constant Propagation

Constant propagation involves the folding and substitution of constant computations throughout a basic block. If the result of a computation is a constant, the instruction is deleted, and the resultant constant is used as an immediate operand in subsequent instructions that reference the original result. Also, if the operands of a conditional branch are constant, the branch can be changed to an unconditional branch or deleted.

Coloring Register Allocation

Many components introduce additional uses of registers or prolong the use of existing registers over larger portions of the procedure. Near-optimal use of the available registers becomes crucial after these optimizations have been made.

Global register allocation based on a method of graph coloring is performed. The register resources are partitioned into groups of disjoint definitions and uses called register webs. Then, using the exposed uses information, interferences between webs are computed. An interference occurs when two webs must be assigned different machine registers. Registers that are copies of each other are assigned to the same register and the copies are eliminated. The webs are sorted based on the number of interfer-

ences each contains. Then register assignment is done using this ordering. When the register allocator runs out of registers, it frees a register by saving another one to memory temporarily. A heuristic algorithm is used to choose which register to save. For example, registers used heavily within a loop will not be saved to free a register.

Peephole Optimizations

The peephole optimizer uses a dictionary of equivalent instruction patterns to simplify instruction sequences. Some of the patterns identify simplifications to addressing mode changes, bit manipulations, and data type conversions.

Branch Optimizations

The branch optimizer component traverses the instructions, transforming branch instruction sequences into more efficient instruction sequences. It converts branches over single instructions to instructions with conditional nullification. A branch whose target is the next instruction is deleted. Branch chains involving both unconditional and conditional branches are combined into shorter sequences wherever possible. For example, a conditional branch to an unconditional branch is changed to a single conditional branch.

Dead Code Elimination

Dead code is code that cannot be reached at program execution, since no branch to it or fall-through exists. This code is deleted.

Scheduler

The instruction scheduler reorders the instructions within a basic block, minimizing load/store and floating-point interlocks. It also schedules the instructions following branches.

Suneel Jain
Development Engineer
Information Technology Group

(continued from page 5)

dent on the register use of the individual procedure. A special calling convention has been adopted to allow some complex operations to be implemented in low-level routines known as *millicode*, which incur little overhead for saving registers and status.

Compiling to a reduced instruction set can be simplified because the compiler need not make complicated choices among a number of instructions that have similar effects. In the new architecture, all arithmetic, logical, or conditional instructions are register-based. All memory access is done through explicit loads and stores. Thus the compiler need not choose among instructions with a multitude of addressing modes. The compiler's task is further simplified by the fact that the instruction set has been constructed in a very symmetrical manner. All instructions are the same length, and there are a limited number of instruction formats. In addition to simplifying the task of code generation, this makes the task of optimization easier as well. The optimizer need not handle transformations between instructions that have widely varying formats and addressing modes. The symmetry of the instruction set makes the tasks of replacing or deleting one or more instructions much easier.

Of course, the reduced instruction set computer, though simplifying some aspects of the compilation, requires more of the compilers in other areas. Having a large number of registers places the burden on the compilers to generate code that can use these registers efficiently. Other aspects of this new architecture also require the compilers to be more intelligent about code generation. For example, the instruction pipeline has become more exposed and, as mentioned earlier, the instruction following a branch may be executed before the branch takes effect. The compiler therefore needs to schedule such instructions effectively. In addition, loads from memory, which also require more than a single cycle, will interlock with the following instruction if the target register is used immediately. The compiler can increase execution speed by scheduling instructions to avoid these interlocks. The optimizer can also improve the effectiveness of a floating-point coprocessor by eliminating unnecessary coprocessor memory accesses and by reordering the floating-point instructions.

In addition to such optimizations, which are designed to exploit specific architectural features, conventional optimizations such as common subexpression elimination, loop invariant code motion, induction variable elaboration, and local constant propagation were also implemented.[3] These have a major impact on the performance of any computer. Such optimizations reduce the frequency of loads, stores, and multiplies, and allow the processor to be used with greater efficiency. However, the favorable cost/performance of the new HP architecture can be realized even without optimization.

The Compiler System

All of the compilers for the new architecture share a common overall design structure. This allows easy integration of common functional components including a symbolic debugger, a code generator, an optimizer, and a linker. This integration was achieved through detailed planning, which involved the participation of engineers across many language products. Of the new compilers, the Fortran/77, Pascal, and COBOL compilers will appear very familiar to some of our customers, since they were developed from existing products available on the HP 3000 family of computers. All of these compilers conform to HP standard specifications for their respective languages, and thus will provide smooth migration from the HP 1000, HP 3000, and HP 9000 product lines. The C compiler is a new product, and as mentioned earlier, was the compiler used to prototype the instruction set from its earliest design phase. The C compiler conforms to recognized industry standard language specifications. Other compilers under development will be integrated into this compiler system.

To achieve successful integration of compilers into a homogeneous compiling system it was necessary to define distinct processing phases and their exact interfaces in terms of data and control transfer. Each compiler begins execution through the *front end*. This includes the lexical, syntactic, and semantic analysis prescribed by each lan-

guage standard. The front ends generate intermediate codes from the source program, and pass these codes to the code generators. The intermediate codes are at a higher level than the machine code generated by a later phase, and allow a certain degree of machine abstraction within the front ends.

Two distinct code generators are used. They provide varying degrees of independence from the front ends. Each interfaces to the front ends through an intermediate code. One of these code generation techniques has already been used in two compiler products for the HP 3000. Fig. 2 shows the overall design of the compilers. Each phase of the compilation process is pictured as it relates to the other phases. The front ends are also responsible for generating data to be used later in the compilation process. For example, the front end generates data concerning source statements and the types, scopes and locations of procedure/function and variable names for later use by the symbolic debugger. In addition, the front end is responsible for the collection of data to be used by the optimizer.

These compilers can be supported by multiple operating systems. The object file format is compatible across operating systems.

Code Generation

The code generators emit machine code into a data structure called SLLIC (Spectrum low-level intermediate code). SLLIC also contains information regarding branches and their targets, and thus provides the foundation for the building of a control flow graph by the optimizer. The SLLIC data structure contains the machine instructions and the specifications for the run-time environment, including the program data space, the literal pool, and data initialization. SLLIC also holds the symbolic debug information generated by the front end, is the medium for later optimization, and is used to create the object file.

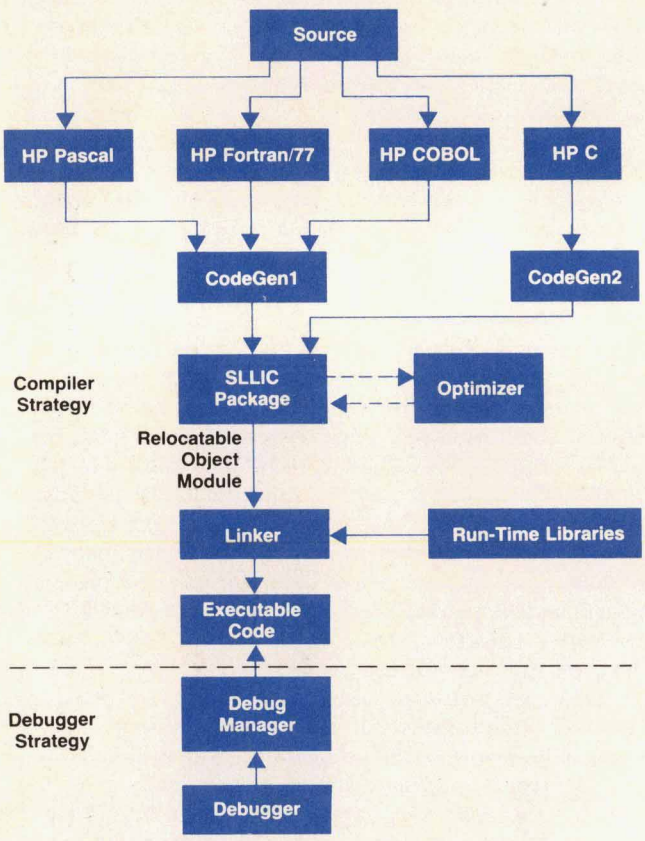


Fig. 2. *The compiler system for HP's new generation of high-precision-architecture computers.*

The reduced instruction set places some extra burden on the code generators when emitting code for high-level language constructs such as byte moves, decimal operations, and procedure calls. Since the instruction set contains no complex instructions to aid in the implementation of these constructs, the code generators are forced to use combinations of the simpler instructions to achieve the same functionality. However, even in complex instruction set architectures, complex case analysis is usually required to use the complex instructions correctly. Since there is little redundancy in the reduced instruction set, most often no choice of alternative instruction sequences exists. The optimizer is the best place for these code sequences to be streamlined, and because of this the overall compiler design is driven by optimization considerations. In particular, the optimizer places restrictions upon the code generators.

The first class of such restrictions involves the presentation of branch instructions. The optimizer requires that all branches initially be followed by a NOP (no operation) instruction. This restriction allows the optimizer to schedule instructions easily to minimize interlocks caused by data and register access. These NOPs are subsequently replaced with useful instructions, or eliminated.

The second class of restrictions concerns register use. Register allocation is performed within the optimizer. Rather than use the actual machine registers, the code generators use symbolic registers chosen from an infinite register set. These symbolic registers are mapped to the set of actual machine registers by the register allocator. Although register *allocation* is the traditional name for such an activity, register *assignment* is more accurate in this context. The code generators are also required to associate every syntactically equivalent expression in each procedure with a unique symbolic register number. The symbolic register number is used by the optimizer to associate each expression with a value number (each run-time *value* has a unique *number*). Value numbering the symbolic registers aids in the detection of common subexpressions within the optimizer. For example, every time the local variable i is loaded it is loaded into the same symbolic register, and every time the same two symbolic registers are added together the result is placed into a symbolic register dedicated to hold that value.

Although the optimizer performs transformations at the machine instruction level, there are occasions where it could benefit from the existence of slightly modified and/or additional instructions. *Pseudoinstructions* are instructions that map to one or more machine instructions and are only valid within the SLLIC data structure as a software convention recognized between the code generators and the optimizer. For example, the NOP instruction mentioned above is actually a pseudoinstruction. No such instruction exists on the machine, although there are many instruction/operand combinations whose net effect would be null. The

NOP pseudoinstruction saves the optimizer from having to recognize all those sequences. Another group of pseudoinstructions has been defined to allow the optimizer to view all the actual machine instructions in the same canonical form, without being restricted by the register use prescribed by the instructions. For example, some instructions use the same register as both a source and a target. This makes optimization very difficult for that instruction. The solution involves the definition of a set of pseudoinstructions, each of which maps to a two-instruction sequence, first to copy the source register to a new symbolic register, and then to perform the operation on that new register. The copy instruction will usually be eliminated by a later phase of the optimizer.

Another class of perhaps more important pseudoinstructions involves the encapsulation of common operations that are traditionally supported directly by hardware, but in a reduced instruction set are only supported through the generation of code sequences. Examples include multiplication, division, and remainder. Rather than have each code generator contain the logic to emit some correct sequence of instructions to perform multiplication, a set of pseudoinstructions has been defined that makes it appear as if a high-level multiplication instruction exists in the architecture. Each of the pseudoinstructions is defined in terms of one register target and either two register operands or one register operand and one immediate. The use of these pseudoinstructions also aids the optimizer in the detection of common subexpressions, loop invariants, and induction variables by reducing the complexity of the code sequences the optimizer must recognize.

Control flow restrictions are also placed on generated code. A *basic block* is defined as a straight-line sequence of code that contains no transfer of control out of or into its midst. If the code generator wishes to set the carry/borrow bit in the status register, it must use that result within the same basic block. Otherwise, the optimizer cannot guarantee its validity. Also, all argument registers for a procedure/function call must be loaded in the same basic block that contains the procedure call. This restriction helps the register allocator by limiting the instances where hard-coded (actual) machine registers can be *live* (active) across basic block boundaries.

Optimization

After the SLLIC data structure has been generated by the code generator, a call is made to the optimizer so that it can begin its processing. The optimizer performs intraprocedural local and global optimizations, and can be turned on and off on a procedure-by-procedure basis by the programmer through the use of compiler options and directives specific to each compiler. Three levels of optimization are supported and can also be selected at the procedural level.

Optimization is implemented at the machine instruction level for two reasons. First, since the throughput of the processor is most affected by the requests made of the memory unit and cache, optimizations that reduce the number of requests made, and optimizations that rearrange these requests to suit the memory unit best, are of the most value. It is only at the machine level that all memory accesses become exposed, and are available candidates for such optimizations. Second, the machine level is the common denominator for all the compilers, and will continue to be for future compilers for the architecture. This allows the implementation of one optimizer for the entire family of compilers. In addition to very machine specific optimizations, a number of theoretically machine independent optimizations (for example, loop optimizations) are also included. These also benefit from their low-level implementation, since all potential candidates are exposed. For example, performing loop optimizations at the machine level allows the optimizer to move constants outside the loop, since the machine has many registers to hold them. In summary, no optimization has been adversely affected by this strategy; instead, there have been only benefits.

Level 0 optimization is intended to be used during program development. It is difficult to support symbolic debugging in the presence of all optimizations, since many optimizations reorder or delete instruction sequences. Nonsymbolic debugging is available for fully optimized programs, but users will still find it easier to debug nonoptimized code since the relationship between the source and object code is clearer. No code transformations are made at level 0 that would preclude the use of a symbolic debugger. In particular, level 0 optimizations include some copy and NOP elimination, and limited branch scheduling. In addition, the components that physically exist as part of the optimizer, but are required to produce an executable program, are invoked. These include register allocation and branch fixing (replacing short branches with long branches where necessary).

After program correctness has been demonstrated using only level 0 optimizations, the programmer can use the more extensive optimization levels. There are two additional levels of optimization, either of which results in code reordering. The level any particular optimization component falls into is dependent upon the type of information it requires to perform correct program transformations. The calculation of data flow information gives the optimizer information regarding all the resources in the program. These resources include general registers, dedicated and status registers, and memory locations (variables). The information gleaned includes where each resource is defined and used within the procedure, and is critical for some optimization algorithms. Level 1 optimizations require no data flow information, therefore adding only a few additional optimizations over level 0. Invoking the optimizer at level 2 will cause all optimizations to be performed. This requires data flow information to be calculated.

Level 1 optimization introduces three new optimizations: peephole and branch optimizations and full instruction scheduling. Peephole optimizations are performed by pattern matching short instruction sequences in the code to corresponding templates in the peephole optimizer. An example of a transformation is seen in the C source expression

```
if (flag & 0x8)
```

which tests to see that the fourth bit from the right is set in the integer flag. The unoptimized code is

```
LDO       8(0), 19      ;  load immediate 8 into r19
AND       31,19,20      ;  intersect r31 (flag) with r19 into r20
COMIBT,=  0,20,label    ;  compare result against 0 and branch
```

Peephole optimization replaces these three instructions with the one instruction

```
BB,>=     31,28,label   ;  branch on bit
```

which will branch if bit 28 (numbered left to right from 0) in r31 (the register containing flag) is equal to 0.

Level 1 optimization also includes a branch optimizer whose task is to eliminate unnecessary branches and some unreachable code. Among other tasks, it replaces branch chains with a single branch, and changes conditional branches whose targets are unconditional branches to a single conditional branch.

The limited instruction scheduling algorithm of level 0 is replaced with a much more thorough component in level 1. Level 0 scheduling is restricted to replacing or removing the NOPs following branches where possible, since code sequence ordering must be preserved for the symbolic debugger. In addition to this, level 1 instructions are scheduled with the goal of minimizing memory interlocks. The following typify the types of transformations made:

- Separate a load from the instruction that uses the loaded register
- Separate store and load instruction sequences
- Separate floating-point instructions from each other to improve throughput of the floating-point unit.

Instruction scheduling is accomplished by first constructing a dependency graph that details data dependencies between instructions. Targeted instructions are separated by data independent instructions discovered in the graph.

The same register allocator is used in level 0 and level 1 optimization. It makes one backwards pass over each procedure to determine where the registers are defined and used and whether or not they are live across a call. It uses this information as a basis for replacing the symbolic registers with actual machine registers. Some copy elimination is also performed by this allocator.

Level 2 optimizations include all level 1 optimizations as well as local constant propagation, local peephole transformations, local redundant definition elimination, common subexpression and redundant load/store elimination, loop invariant code motion, induction variable elaboration and strength reduction, and another register allocator. The register allocator used in level 2 is partially based on graph coloring technology.[4] Fully optimized code contains many more live registers than partially optimized or nonoptimized code. This register allocator handles many live registers better than the register allocator of levels 0 and 1. It has access to the data flow information calculated for the symbolic registers and information regarding the frequency of execution for each basic block.

Control Flow and Data Flow Analysis

All of the optimizations introduced in level 2 require data flow information. In addition, a certain amount of control flow information is required to do loop-based optimizations. Data flow analysis provides information to the optimizer about the pattern of definition and use of each resource. For each basic block in the program, data flow information indicates what definitions may reach the block (reaching definitions) and what later uses may be affected by local definitions (exposed uses). Control flow information in the optimizer is contained in the basic block and interval structures. *Basic block analysis* identifies blocks of code that have no internal branching. *Interval analysis* identifies patterns of control flow such as if-then-else and loop constructs.[5] Intervals simplify data flow calculations, identify loops for the loop-based optimizations, and enable partial update of data flow information.

In the optimizer, control flow analysis and data flow analysis are performed in concert. First, basic blocks are identified. Second, local data flow information is calculated for each basic block. Third, interval analysis exposes the structure of the program. Finally, using the interval structure as a basis for its calculation rules, global data flow analysis calculates the reaching definitions and exposed uses.

Basic block analysis of the SLLIC data structure results in a graph structure where each basic block identifies a sequence of instructions, along with the predecessor and successor basic blocks. The interval structure is built on top of this, with the smallest interval being a basic block. Intervals other than basic blocks contain subintervals which may themselves be any type of interval. Interval types include basic block, sequential block (the subintervals follow each other in sequential order), if-then, if-then-else, self loop, while loop, repeat loop, and switch (case statement). When no such interval is recognized, a set of subintervals may be contained in either a *proper interval*

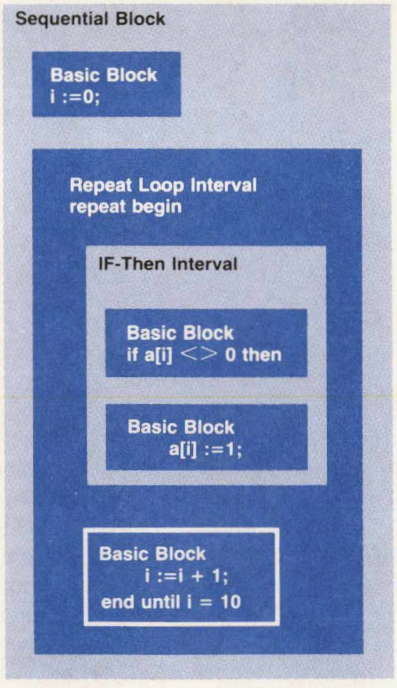


Fig. 3. *This figure illustrates the interval structure of a simple sequence of Pascal code. The nested boxes represent the interval hierarchy.*

(if the control flow is well-behaved) or an *improper interval* (if it contains multiple-entry cycles or targets of unknown branches). An entire procedure will be represented by a single interval with multiple descendants. Fig. 3 shows the interval structure for a simple Pascal program.

Calculation of data flow information begins with an analysis of what resources are used and defined by each basic block. Each use or definition of a resource is identified by a unique *sequence number*. Associated with each sequence number is information regarding what resource is being referenced, and whether it is a use or a definition. Each SLLIC instruction entry contains sequence numbers for all of the resources defined or used by that instruction. The local data flow analysis determines what local uses are exposed at the top of the basic block (i.e., there is a use of a resource with no preceding definition in that block) and what local definitions will reach the end of the block (i.e., they define a resource that is not redefined later in the block). The local data flow analysis makes a forward and backward pass through the instructions in a basic block to determine this information.

Local data flow information is propagated out from the basic blocks to the outermost interval. Then, information about reaching definitions and exposed uses is propagated inward to the basic block level. For known interval types, this involves a straightforward calculation for each subinterval. For proper intervals, this calculation must be performed twice for each subinterval, and for improper intervals, the number of passes is limited by the number of subintervals.

As each component of the optimizer makes transformations to the SLLIC graph, the data flow information becomes inaccurate. Two strategies are employed to bring this information up-to-date: *patching* of the existing data flow information and partial recalculation. For all optimizations except induction variable elimination, the data flow information can be patched by using information about the nature of the transformation to determine exactly how the data flow information must be changed. All transformations take place within the loop interval in induction variable elimination. The update of data flow information within the loop is performed by recalculating the local data flow information where a change has been made, and then by propagating that change out to the loop interval. The effect of induction variable elimination on intervals external to the loop is limited, and this update is performed by patching the data flow information for these intervals.

Aliasing

The concept of resources has already been presented in the earlier discussion of data flow analysis. The optimizer provides a component called the *resource manager* for use throughout the compiler phases. The resource manager is responsible for the maintenance of information regarding the numbers and types of resources within each procedure. For example, when the code generator needs a new symbolic register, it asks the resource manager for one. The front ends also allocate resources corresponding to memory locations for every variable in each procedure. The resources allocated by the resource manager are called *resource numbers*. The role of the resource manager is especially important in this family of compilers. It provides a way for the front end, which deals with memory resources in terms of programmer variable names, and the optimizer, which deals with memory resources in terms of actual memory locations, to communicate the relationship between the two.

The most basic use of the resource numbers obtained through the resource manager is the identification of unique programmer variables. The SLLIC instructions are decorated with information that associates resource numbers with each operand. This allows the optimizer to recognize uses of the same variable without having to compare addresses. The necessity for communication between the front ends and the optimizer is demonstrated by the following simplified example of C source code:

```
proc() {
  int i, j, k, *p;
   .
   .
   .
i = j + k;
*p = 1;
i = j + k;
   .
   .
   .
}
```

At first glance it might seem that the second calculation of j + k is redundant, and in fact it is a common subexpression that need only be calculated once. However, if the pointer p has been set previously to point to either j or k, then the statement *p = 1 might change the value of either j or k. If p has been assigned to point to j, then we say that *p and j are *aliased* to each other. Every front end includes a component called a *gatherer*[6] whose responsibility it is to collect information concerning the ways in which memory resources in each procedure relate to each other. This information is cast in terms of resource numbers, and is collected in a similar manner by each front end. Each gatherer applies a set of language specific alias rules to the source. A later component of the optimizer called the aliaser reorganizes this information in terms more suitable for use by the local data flow component of the optimizer.

Each gatherer had to solve aliasing problems specific to its particular target language. For example, the Pascal gatherer was able to use Pascal's strong typing to aid in building sets of resources that a pointer of some particular type can point to. Since C does not have strong typing, the C gatherer could make no such assumptions. The COBOL compiler had to solve the aliasing problems that are introduced with the REDEFINE statement, which can make data items look like arrays. Fig. 4 shows the structure of the new compilers from an aliasing perspective. It details data and control dependencies. Once the aliasing data has been incorporated into the data flow information, every component in the optimizer has access to the information, and incorrect program transformations are prevented.

The aliaser also finishes the calculation of the aliasing

relationships by calculating the transitive closure* on the aliasing information collected by the gatherers. The need for this calculation is seen in the following skeleton Pascal example:

```
procedure p;
 begin
 p : ^integer;
 q : ^integer;
  .
  .
  .
 p := q;
  .
  .
  .
 q := p;
  .
  .
  .
 end;
```

The aliasing information concerning q must be transferred to p, and vice versa, because of the effects of the two assignment statments shown. The aliaser is an optimizer component used by all the front ends, and requires no language specific data. Another type of memory aliasing occurs when two or more programmer variables can overlap with one another in memory. This happens within C unions and Fortran equivalence statements. Each gatherer must also deal with this issue, as well as collecting information concerning the side effects of procedure and function calls and the use of arrays.

*Transitive closure: For a given resource, the set of resources that can be shown to be aliased to the given resource by any sequence of aliasing relationships.

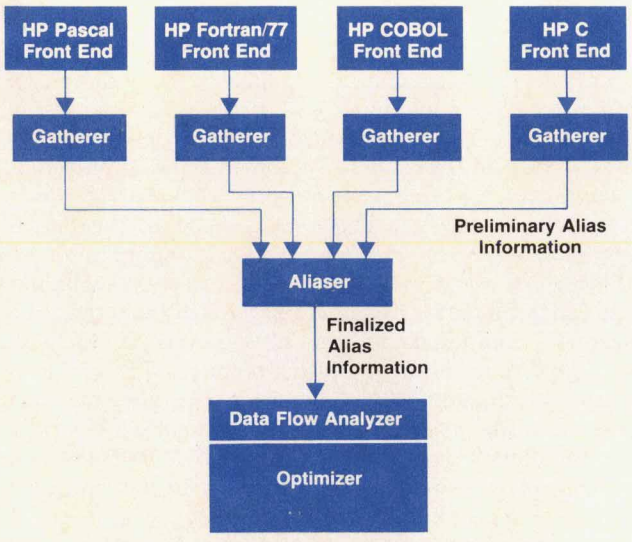


Fig. 4. *Scheme for the collection of alias information.*

The SLLIC Package

The SLLIC data structure is allocated, maintained, and manipulated by a collection of routines called the *SLLIC package*. Each code generator is required to use these routines. The SLLIC package produces an object file from the SLLIC graph it is presented with, which is either optimized or unoptimized. During implementation it was relatively easy to experiment with the design of the object file, since its creation is only implemented in one place. The object file is designed to be transportable between multiple operating systems running on the same architecture.

The SLLIC graph also contains the symbolic debug information produced by the front end. This information is placed into the object file by the SLLIC package. The last step in the compilation process is the link phase. The linker is designed to support multiple operating systems. As much as possible, our goal has been for the new compilers to remain unchanged across operating systems, an invaluable characteristic for application development.

Addressing RISC Myths

The new compiling system provides a language development system that is consistent across languages. However, each language presents unique requirements to this system. Mapping high-level language constructs to a reduced-complexity computer requires the development of new implementation strategies. Procedure calls, multiplication, and other complex operations often implemented in microcode or supported in the hardware can be addressed with code sequences tuned to the specific need. The following discussion is presented in terms of several misconceptions, or myths, that have appeared in speculative discussions concerning code generation for reduced-complexity architectures. Each myth is followed by a description of the approach adopted for the new HP compilers.

Myth: An architected procedure call instruction is necessary for efficient procedure calls.

Modern programming technique encourages programmers to write small, well-structured procedures rather than large monolithic routines. This tends to increase the frequency of procedure calls, thus making procedure call efficiency crucial to overall system performance.

Many machines, like the HP 3000, provide instructions to perform most of the steps that make up a procedure call. The new HP high-precision architecture does not. The mechanism of a procedure call is not architected, but instead is accomplished by a software convention using the simple hardwired instructions. This provides more flexibility in procedure calls and ultimately a more efficient call mechanism.

Procedure calls are more than just a branch and return in the flow of control. The procedure call mechanism must also provide for the passing of parameters, the saving of the caller's environment, and the establishment of an environment for the called procedure. The procedure return mechanism must provide for the restoration of the calling procedure's environment and the saving of return values.

The new HP machines are register-based machines, but

by convention a stack is provided for data storage. The most straightforward approach to procedure calls on these machines assumes that the calling procedure acquires the responsibility for preserving its state. This approach employs the following steps:

- Save all registers whose contents must be preserved across the procedure call. This prevents the called procedure, which will also use and modify registers, from affecting the calling procedure's state. On return, those register values are restored.
- Evaluate parameters in order and push them onto the stack. This makes them available to the called procedure which, by convention, knows how to access them.
- Push a *frame marker*. This is a fixed-size area containing several pieces of information. Among these is the *static link*, which provides information needed by the called procedure to address the local variables and parameters of the calling procedure. The return address of the calling procedure is also found in the stack marker.
- Branch to the entry point of the called procedure.

To return from the call, the called procedure extracts the return address from the stack marker and branches to it. The calling procedure then removes the parameters from the stack and restores all saved registers before program flow continues.

This simple model correctly implements the steps needed to execute a procedure call, but is relatively expensive. The model forces the caller to assume all responsibility for preserving its state. This is a safe approach, but causes too many register saves to occur. To optimize the program's execution, the compiler makes extensive use of registers to hold local variables and temporary values. These registers must be saved at a procedure call and restored at the return. The model also has a high overhead incurred by the loading and storing of parameters and linkage information. The ultimate goal of the procedure call convention is to reduce the cost of a call by reducing memory accesses.

The new compilers minimize this problem by introducing a procedure call convention that includes a register partition. The registers are partitioned into *caller-saves* (the calling procedure is responsible for saving and restoring them), *callee-saves* (the called procedure must save them at entry and restore them at exit), and *linkage* registers. Thirteen of the 32 registers are in the caller-saves partition and 16 are in the callee-saves partition. This spreads the responsibility for saving registers between the calling and called procedures and leaves some registers available for linkage.

The register allocator avoids unnecessary register saves by using caller-saves registers for values that need not be preserved. Values that must be saved are placed into registers from the callee-saves partition. At procedure entry, only those callee-saves registers used in the procedure are saved. This minimizes the number of loads and stores of registers during the course of a call. The partition of registers is not inflexible; if more registers are needed from a particular partition than are available, registers can be borrowed from the other partition. The penalty for using these additional registers is that they must be saved and restored, but this overhead is incurred only when many registers are needed, not for all calls.

In the simple model, all parameters are passed by being placed on the stack. This is expensive because memory references are made to push each parameter and as a consequence the stack size is constantly altered. The new compilers allocate a permanent parameter area large enough to hold the parameters for all calls performed by the procedure. They also minimize memory references when storing parameters by using a combination of registers and memory to pass parameters. Four registers from the callee-saves partition are used to pass user parameters; each holds a single 32-bit value or half of a 64-bit value. Since procedures frequently have few parameters, the four registers are usually enough to contain them all. This removes the necessity of storing parameter values in the parameter area before the call. If more than four 32-bit parameters are passed, the additional ones are stored in the preallocated parameter area. If a parameter is larger than 64 bits, its address is passed and the called procedure copies it to a temporary area.

Additional savings on stores and loads occur when the called procedure is a leaf routine. As mentioned previously, the optimizer attempts to maximize the use of registers to hold variable values. When a procedure is a leaf, the register allocator uses the caller-saves registers for this purpose, thus eliminating register saves for both the calling and called procedures. It is never necessary to store the return address or parameter registers of a leaf routine since they will not be modified by subsequent calls.

Leaf routines do not need to build a stack frame, since they make no procedure calls. Also, if the allocator succeeds in representing all local variables as registers, it is not necessary to build the local variable area at entry to the leaf procedure.

The convention prescribes other uses of registers to eliminate other loads and stores at procedure calls. The return address is always stored in a particular register, as is the static link if it is needed.

To summarize, the procedure call convention used in the new HP computers streamlines the overhead of procedure calls by minimizing the number of memory references. Maximal use of registers is made to limit the number of memory accesses needed to handle parameters and linkage. Similarly, the convention minimizes the need to store values contained in registers and does not interfere with attempts at optimization.

Myth: The simple instructions available in RISC result in significant code expansion.

Many applications, especially commercial applications, assume the existence of complex high-level instructions typically implemented by the system architecture in microcode or hardware. Detractors of RISC argue that significant code expansion is unavoidable since the architecture lacks these instructions. Early results do not substantiate this argument.[7,8] The new HP architecture does not provide complex instructions because of their impact on overall system performance and cost, but their functionality is available through other means.

As described in an earlier article,[2] the new HP machines

do not have a microcoded architecture and all of the instructions are implemented in hardware. The instructions on microcoded machines are implemented in two ways. At the basic level, instructions are realized in hardware. More complex instructions are then produced by writing subroutines of these hardware instructions. Collectively, these constitute the microcode of the machine. Which instructions are in hardware and which are in microcode are determined by the performance and cost goals for the system. Since HP's reduced instruction set is implemented solely at the hardware level, subroutines of instructions are equivalent to the microcode in conventional architectures.

To provide the functionality of the complex instructions usually found in the architecture of conventional machines, the design team developed the alternative concept of *millicode instructions* or routines. Millicode is HP's implementation of complex instructions using the simple hardware instructions packaged into subroutines. Millicode serves the same purpose as traditional microcode, but is common across all machines of the family rather than specific to each.

The advantages of implementing functionality as millicode are many. Microcoded machines may contain hidden performance penalties on all instructions to support multiple levels of instruction implementation. This is not the case for millicode. From an architectural viewpoint, millicode is just a collection of subroutines indistinguishable from other subroutines. A millicode instruction is executed by calling the appropriate millicode subroutine. Thus, the expense of executing a millicode instruction is only present when the instruction is used. The addition of millicode instructions has no hardware cost and hence no direct influence on system cost. It is relatively easy and inexpensive to upgrade or modify millicode in the field, and it can continue to be improved, extended, and tuned over time.

Unlike most microcode, millicode can be written in the same high-level languages as other applications, reducing development costs yet still allowing for optimization of the resultant code. Severely performance-critical millicode can still be assembly level coded in instances where the performance gain over compiled code is justified. The size of millicode instructions and the number of such instructions are not constrained by considerations of the size of available control store. Millicode resides in the system as subroutines in normally managed memory, either in virtual memory where it can be paged into and out of the system as needed, or in resident memory as performance considerations dictate. A consequence of not being bound by restrictive space considerations is that compiler writers are free to create many more specialized instructions in millicode than would be possible in a microcoded architecture, and thus are able to create more optimal solutions for specific situations.

Most fixed instruction sets contain complex instructions that are overly general. This is necessary since it is costly to architect many variations of an instruction. Examples of this are the MVB (move bytes) and MVW (move words) instructions on the HP 3000. They are capable of moving any number of items from any arbitrary source location to any target location. Yet, the compiler's code generators frequently have more information available about the operands of these instructions that could be used to advantage if other instructions were available. The code generators frequently know whether the operands overlap, whether the operands are aligned favorably, and the number of items to be moved. On microcoded machines, this information is lost after code generation and must be recreated by the microcode during each execution of the instruction. On the new HP computers, the code generators can apply such information to select a specialized millicode instruction that will produce a faster run-time execution of the operation than would be possible for a generalized routine.

Access to millicode instructions is through a mechanism similar to a procedure call. However, additional restrictions placed on the implementation of millicode routines prevent the introduction of any barriers to optimization. Millicode routines must be leaf routines and must have no effect on any registers or memory locations other than the operands and a few scratch registers. Since millicode calls are represented in SLLIC as pseudoinstructions, the optimizer can readily distinguish millicode calls from procedure calls. Millicode calls also use different linkage registers from procedure calls, so there is no necessity of preserving the procedure's linkage registers before invoking millicode instructions.

The only disadvantage of the millicode approach over microcode is that the initiation of a millicode instruction involves an overhead of at least two instructions. Even so, it is important to realize that for most applications, millicode instructions are infrequently needed, and their overhead is incurred only when they are used. The high-precision architecture provides the frequently needed instructions directly in hardware.

Myth: RISC machines must implement integer multiplication as successive additions.

Integer multiplication is frequently an architected instruction. The new architecture has no such instruction but provides others that support an effective implementation of multiplication. It also provides for inclusion of a high-speed hardware multiplier in a special function unit.[2]

Our measurements reveal that most multiplication operations generated by user programs involve multiplications by small constants. Many of these occurrences are explicitly in the source code, but many more are introduced by the compiler for address and array reference evaluation. The new compilers have available a trio of instructions that perform shift and add functions in a single cycle. These instructions, SH1ADD (shift left once and add), SH2ADD (shift left twice and add) and SH3ADD (shift left three times and add) can be combined in sequences to perform multiplication by constants in very few instructions. Multiplications by most constants with absolute values less than 1040 can be accomplished in fewer than five cycles. Negatively signed constants require an additional instruction to apply the sign to the result. Multiplication by all constants that are exact powers of 2 can be performed with a single shift instruction unless overflow conditions are to be detected. Additionally, multiplications by 4 or 2 for indexed address-

ing can be avoided entirely. The LDWX (load word indexed) and LDHX (load half-word indexed) instructions optionally perform unit indexing, which combines multiplication of the index value with the address computation in the hardware.

The following examples illustrate multiplication by various small constants.

```
Source code:
        4*k
Assembly code:
        SH2ADD    8,0,9          ; shift r8 (k) left 2 places,
                                   add to r0 (zero) into r9
Source code:
        -163*k
Assembly code:
        SH3ADD    8,8,1          ; shift r8 (k) left 3 places, add
                                   to itself into r1
        SH3ADD    1,1,1          ; shift r1 left 3 places, add to
                                   itself into r1
        SH1ADD    1,8,1          ; shift r1 left 1 place, add to
                                   k into r1
        SUB       0,1,1          ; subtract result from 0 to
                                   negate; back into r1
Source code:
        A(k)
Assembly code:
        LDO       -404(30),9     ; load array base address
                                   into r9
        LDW       -56(0,30),7    ; load unit index value into r7
        LDWX,S    7(0,9),5       ; multiply index by 4 and
                                   load element into r5
```

When neither operand is constant or if the constant is such that the in-line code sequence would be too large, integer multiplication is accomplished with a millicode instruction. The multiply millicode instruction operates under the premise that even when the operands are unknown at compile time, one of them is still likely to be a small value. Application of this to the multiplication algorithm yields an average multiplication time of 20 cycles,which is comparable to an iterative hardware implementation.

Myth: RISC machines cannot support commercial applications languages.

A popular myth about RISC architectures is that they cannot effectively support languages like COBOL. This belief is based on the premise that RISC architectures cannot provide hardware support for the constructs and data types of COBOL-like languages while maintaining the one-instruction-one-cycle advantages of RISC. As a consequence, some feel that the code expansion resulting from performing COBOL operations using only the simple architected instructions would be prohibitive. The significance of this is often overstated. Instruction traces of COBOL programs measured on the HP 3000 indicate that the frequency of decimal arithmetic instructions is very low. This is because much of the COBOL program's execution time is spent in the operating system and other subsystems.

COBOL does place demands on machine architects and compiler designers that are different from those of languages like C, Fortran, and Pascal. The data items provided in the latter languages are represented in binary and hence are native to the host machine. COBOL data types also include packed and unpacked decimal, which are not commonly native and must be supported in ways other than directly in hardware.

The usual solution on conventional machines is to provide a commercial instruction set in microcode. These additional instructions include those that perform COBOL field (variable) moves, arithmetic for packed decimal values, alignment, and conversions between the various arithmetic types.

In the new HP machines, millicode instructions are used to provide the functionality of a microcoded commercial instruction set. This allows the encapsulation of COBOL operations while removing the possibility of runaway code expansion. Many COBOL millicode instructions are available to do each class of operation. The compiler expends considerable effort to select the optimal millicode operation based on compile-time information about the operation and its operands. For example, to generate code to perform a COBOL field move, the compiler may consider the operand's relative and absolute field sizes and whether blank or zero padding is needed before selecting the appropriate millicode instruction.

Hardware instructions that assist in the performance of some COBOL operations are architected. These instructions execute in one cycle but perform operations that would otherwise require several instructions. They are emitted by the compiler in in-line code where appropriate and are also used to implement some of the millicode instructions. For example, the DCOR (decimal correct) and UADDCM (unit add complement) instructions allow packed decimal addition to be performed using the binary ADD instruction. UADDCM prepares an operand for addition and the DCOR restores the result to packed decimal form after the addition. For example:

```
r1 and r2 contain packed decimal operands
r3 contains the constant X'99999999'

UADDCM 1,3,31    ; pre-bias operand into r31
ADD    2,31,31   ; perform binary add
DCOR   31,31     ; correct result
```

Millicode instructions support arithmetic for both packed and unpacked decimal data. This is a departure from the HP 3000, since on that machine unpacked arithmetic is performed by first converting the operand to packed format, performing the arithmetic operation on the packed data, and then converting the result back to unpacked representation. Operations occur frequently enough on unpacked data to justify the implementation of unpacked arithmetic routines. The additional cost to implement them is minimal and avoids the overhead of converting operands between the two types. An example of the code to perform an unpacked decimal add is:

r1 and r2 contain unpacked decimal operands
r3 contains the constant X'96969696'
r4 contains the constant X'0f0f0f0f'
r5 contains the constant X'30303030'

```
ADD     3,1,31      ; pre-bias operand into r31
ADD     31,2,31     ; binary add into r31
DCOR    31,31       ; correct result
AND     4,31,31     ; mask result
OR      5,31,31     ; restore sum to unpacked decimal
```

In summary, COBOL is supported with a blend of hardware assist instructions and millicode instructions. The compiled code is compact and meets the run-time execution performance goals.

Conclusions

The Spectrum program began as a joint effort of hardware and software engineers. This early communication allowed high-level language issues to be addressed in the architectural design.

The new HP compiling system was designed with a reduced-complexity machine in mind. Register allocation, instruction scheduling, and traditional optimizations allow compiled programs to make efficient use of registers and low-level instructions.

Early measurements have shown that this compiler technology has been successful in exploiting the capabilities of the new architecture. The run-time performance of compiled code consistently meets performance objectives. Compiled code sizes for high-level languages implemented

An Optimization Example

This example illustrates the code generated for the following C program for both the unoptimized and the optimized case.

```
test ( )
{
int i, j;
int a1[25], a2[25], r[25][25];

for (i = 0; i < 25; i++) {
    for (j = 0; j < 25; j++) {
        r [i] [j] = a1 [i] * a2 [j];
    }
  }
}
```

In the example code that follows, the following mnemonics are used:

rp	return pointer, containing the address to which control should be returned upon completion of the procedure
arg0	first parameter register
arg1	second parameter register
sp	stack pointer, pointing to the top of the current frame
mret0	millicode return register
mrp	millicode return pointer.

The value of register zero (r0) is always zero.

The following is a brief description of the instructions used:

LDO	immed(r1),r2	r2←r1 + immed.
LDW	immed(r1),r2	r2←*(r1 + immed)
LDWX,S	r1(r2),r3	r3←*(4*r1 + r2)
STW	r1,immed(r2)	*(r2 + immed)←r1
STWS	r1,immed(r2)	*(r2 + immed)←r1
STWM	r1,immed(r2)	*(r2 + immed)←r1 AND r2←r2 + immed
COMB,<=	r1,r2,label	if r1 <= r2, branch to label
BL	label,r1	branch to label, and put return address into r1 (for procedure call)
BV	0(r1)	branch to address in r1 (for procedure return)
ADD	r1,r2,r3	r3←r1 + r2
SH1ADD	r1,r2,r3	r3←2*r1 + r2
SH2ADD	r1,r2,r3	r3←4*r1 + r2
SH3ADD	r1,r2,r3	r3←8*r1 + r2
COPY	r1,r2	r2←r1
NOP		no effect

In the following step-by-step discussion, the unoptimized code on the left is printed in black, and the optimized code on the right is printed in color. The code appears in its entirety, and can be read from the top down in each column.

Save callee-saves registers and increment stack pointer. Unoptimized case uses no register that needs to be live across a call.

```
LDO     2760(sp),sp     STW     2,-20(0,sp)
                        STWM    3,2768(0,sp)
                        STW     4,-2764(0,sp)
```

Assign zero to i. In the optimized case, i resides in register 19.

```
STW     0,-52(0,sp)     COPY    0,19
```

Compare i to 25. This test is eliminated in the optimized case since the value of i is known.

```
LDW         -52(0,sp),1
LDO         25(0),31
COMB,<=,N   31,1,L2
```

In the optimized version, a number of expressions have been moved out of the loop:

```
{maximum value of j}        LDO     25(0),20
{address of a1}             LDO     -156(sp),22
{address of a2}             LDO     -256(sp),24
{address of r}              LDO     -2756(sp),28
{initial value of 100*i}    LDO     0(0),4
{maximum value of 100*i}    LDO     2500(0),2
```

Initialize j to zero, and compare j to 25. This test has also been eliminated in the optimized version, since the value of j is known. Note that j now resides in register 21.

```
L3
  STW       0,-56(0,sp)     COPY    0,21
  LDW       -56(0,sp),19
```

in this low-level instruction set are comparable to those for more conventional architectures. Use of millicode instructions helped achieve this result. Complex high-level language operations such as procedure calls, multiplication, and COBOL constructs have been implemented efficiently with the low-level instructions provided by the high-precision architecture. A later paper will present performance measurements.

Acknowledgments

The ideas and results presented in this paper are the culmination of the work of many talented engineers involved with the Spectrum compiler program. We would like to acknowledge the individuals who made significant technical contributions to the work presented in this paper in the following areas: early compiler development and optimizer investigation at HP Laboratories, optimizer development, aliasing design and implementation in the compiler front ends, code generator design and implementation, procedure call convention design, and object module specification.

Megan Adams
Robert Ballance
Bruce Blinn
William Buzbee
Don Cameron
Peter Canning
Paul Chan
Cary Coutant
Eric Eidt
Phil Gibbons
Adiel Gorel
Richard Holman
Mike Huey
Audrey Ishizaki
Suneel Jain
Mark Scott Johnson


```
LDO          25(0),20
COMB,<=,N    20,19,L1
```

In the optimized version, the load of a1[i] is moved out of the inner loop, since the value of i is constant in the inner loop.

```
                                    LDWX,S    19(0,22),23
```

Register 28 contains the address of r, and register 4 contains the value 100*i, which is the offset of the ith row of array r. This is constant over the inner loop, and has been moved out.

```
                                    ADD       28,4,3
L6
```

The loop begins with the load of a1[i] into the first parameter register. This value has already been loaded in the optimized version, and need only be copied.

```
LDO          -156(sp),21
LDW          -52(0,sp),22
LDWX,S       22(0,21),arg0      COPY      23,arg0
```

The value of a2[j] is loaded into the second parameter register, and the multiply millicode instruction is called. In the optimized case, the address of a2[0] and the value of j are both already in registers.

```
LDO          -256(sp),1
LDW          -56(0,sp),19
BL           mull,mrp           BL        mull,mrp
LDWX,S       19(0,1),arg1       LDWX,S    21(0,24),arg1
```

Store the result into r[i][j]. The three SHxADD instructions calculate 100*i. Note that most of the following is loop invariant, and has been moved out of the loop in the optimized case.

```
LDO          -2756(sp),19 {address of r}
LDW          -52(0,sp),20 {value of i}
SH1ADD       20,20,21     {r21 ← 3*i}
SH3ADD       21,20,22     {r22 ← 25*i}
SH2ADD       22,0,1       {r1 ← 100*i}
ADD          19,1,31      {address of r + 100*i}
LDW          -56(0,sp),19 {value of j}
SH2ADD       19,31,20     {add j*4 to address}  SH2ADD  21,3,31
STWS         mret0,0(0,20) {store}              STWS    mret0,0(0,31)
```

Increment j.

```
LDW          -56(0,sp),21                      LDO     1(21),21
LDO          1(21),22
STW          22,-56(0,sp)
```

Compare j to the value 25 (already in register 20 in the optimized version). The position after the conditional branch contains no useful instruction in the unoptimized case. In the optimized version, the first instruction of the loop has been copied to this position, and the target adjusted to the following instruction. Because the branch has the nullification flag set (,N), the following instruction will not be executed when the branch is not taken.

```
LDW          -56(0,sp),1
LDO          25(0),31
COMBF,<=     31,1,L6                COMBF,<=,N  20,21,L6+4
NOP                                 LDWX,S      21(0,24),25
L1
```

Increment i, and test for the end of the loop. In the optimized version, induction variable elaboration has removed the 100*i multiplication, and added a new induction variable to contain that value. This value, in register 4, is now tested against a maximum value of 2500, contained in register 2. This branch has been scheduled like the previous branch.

```
LDW          -52(0,sp),19
LDO          1(19),20               LDO       1(19),19
STW          20,-52(0,sp)           LDO       100(4),4
LDW          -52(0,sp),21
LDO          25(0),22
COMBF,<=     22,21,L3               COMBF,<=,N 2,4,L3+4
NOP                                 COPY      0,21
L2
```

Finally, the registers are restored, and control is returned to the calling procedure.

```
                                    LDW       -2788(0,sp),2
                                    LDW       -2764(0,sp),4
BV           0(rp)                  BV        0(rp)
LDO          -2760(sp),sp           LDWM      -2768(0,sp),3
```

Steven Kusmer
Tom Lee
Steve Lilker
Daniel Magenheimer
Tom McNeal
Sue Meloy
Terrence Miller
Angela Morgan
Steve Muchnick
Karl Pettis
David Rickel
Michelle Ruscetta
Steven Saunders
Carolyn Sims
Ron Smith
Kevin Wallace
Alexand Wu

We feel privileged to have the opportunity to present their work. We would like to extend special thanks to Bill Buzbee for his help in providing code examples, and to Suneel Jain for providing the description of the optimization components.

References

1. D.A. Patterson, "Reduced Instruction Set Computers," *Communications of the ACM*, Vol. 28, no. 1, January 1985, pp. 8-21.
2. J.S. Birnbaum and W.S. Worley, Jr., "Beyond RISC: High-Precision Architecture," *Hewlett-Packard Journal*, Vol. 36, no. 8, August 1985, pp. 4-10.
3. A.V. Aho and J.D. Ullman, *Principles of Compiler Design*, Addison-Wesley, 1977.
4. G.J. Chaitin, "Register Allocation and Spilling via Graph Coloring," *Proceedings of the SIGPLAN Symposium on Compiler Construction*, June 1982, pp. 98-105.
5. M. Sharir, "Structural Analysis: A New Approach To Flow Analysis in Optimizing Compilers," *Computer Languages*, Vol. 5, Pergamon Press Ltd., 1980.
6. D.S. Coutant, "Retargetable High-Level Alias Analysis," *Conference Record of the 13th ACM Symposium on Principles of Programming Languages*, January 1986.
7. J.A. Otto, "Predicting Potential COBOL Performance on Low-Level Machine Architectures," *SIGPLAN Notices*, Vol. 20, no. 10, October 1985, pp. 72-78.
8. G. Radin, "The 801 Computer," *Symposium on Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 39-47.

Authors

January 1986

4 Compilers

Jon W. Kelley

With HP since 1975, Jon Kelley has worked on BASIC and RPG compilers for the HP 300 Business Computer and on a prototype optimizer. He has also contributed to the development of code generators for HP 3000 Computers and for the Spectrum program. Jon graduated in 1974 from the University of California at Berkeley with a BA degree in computer science. He lives in Sunnyvale, California and lists fly-fishing, hunting, and flying as outside interests.

Deborah S. Coutant

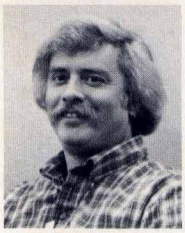
Debbie Coutant earned a BA degree in psychology from the University of Arizona in 1977 and an MS degree in computer science from the University of Arizona in 1981. After joining HP's Information Networks Division in 1981, she worked on Pascal for HP 3000 Computers and later investigated compiler optimization techniques and contributed to the development of code generators and optimizers for the Spectrum program. She is the author of a paper on retargetable alias analysis and is a member of the ACM and SIGPLAN. Born in Bethpage, New York, Debbie lives in San Jose, California. She's married and enjoys playing the French horn in community orchestras. Her other outside interests include racquetball and camping.

Carol L. Hammond

With HP since 1982, Carol Hammond manages an optimizer project in the computer language laboratory of HP's Information Technology Group. In earlier assignments at HP Laboratories she wrote architecture verification programs and worked on a compiler project. She is a member of ACM and SIGPLAN. Carol was born in Long Branch, New Jersey and studied physics at the University of California at Davis (BS 1977). She worked as a professional musician for four years before resuming her studies at the University of California at Berkeley, completing work for an MS degree in computer science in 1983. She lives in San Jose, California and still enjoys singing and playing the piano.

20 Measurement Plotting System

Thomas H. Daniels



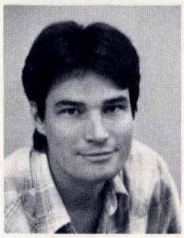
With HP since 1963, Tom Daniels was project manager for the HP 7090A Measurement Plotting System and earlier was project manager for the HP 9872A Plotter. He coauthored an HP Journal article on the HP 9872A. Tom was born in Los Angeles, California and received a BSEE degree from Northrop University in 1963. He's now a resident of Escondido, California, is married, and has two children. His outside interests include woodworking and restoring old Chevrolet Vegas.

John Fenoglio

An R&D manager at HP's San Diego Division, John Fenoglio has been with the company since 1972. He was section manager for the HP 7090A Measurement Plotting System and has held various engineering, marketing, and management positions for analog and digital products. He was born in Texas and holds a BSEE degree from California State Polytechnic University (1972) and an MSEE degree from San Diego State University (1974). He is also the coauthor of an HP Journal article on the HP 7225A Plotter. John is a resident of San Diego, California and an adventure-seeker. He is a skier and scuba diver and has flown in exotic locations, from the polar ice caps to South American jungles. He also restores sports cars and is an amateur radio operator. He has bounced signals off the moon using a 40-foot dish antenna.

24 Measurement Graphics

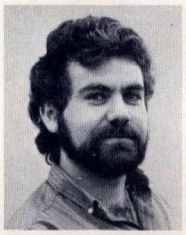
Steven T. Van Voorhis



Steve Van Voorhis holds a BA degree in biology and psychology from the University of California at San Diego (1973) and an MA degree in electrical engineering from San Diego State University (1980). With HP since 1981, he's a project leader in the design graphics section of HP's San Diego Division and was a design engineer on the HP 7090A Measurement Plotting System. He was also a research assistant and design engineer at the University of California at San Diego and is the coauthor of three papers on human visual and auditory responses. A native of California, he was born in Los Angeles and now lives in Solana Beach. He and his wife and two sons enjoy spending time at the beach. He is also a soccer player and is remodeling his house.

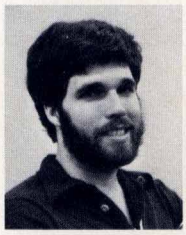
27 Software

Emil Maghakian



With HP since 1979, Emil Maghakian has worked on HP 7310 Printer firmware, on ink-jet printer technology, and on software for various products, including the HP 7090A Measurement Plotting System. He was born in Tehran, Iran and studied computer science at Virginia Polytechnic Institute and State University, receiving his BS degree in 1976 and MS degree in 1978. He was an instructor at Hollins College before coming to HP. His professional interests include computer graphics and man-machine interface problems. A resident of Escondido, California, Emil is married and has two children. He's active in various Armenian organizations in San Diego and is interested in public speaking. He also enjoys soccer and aerobics.

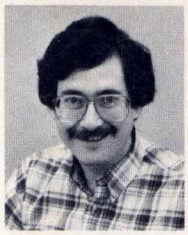
Francis E. Bockman



A computer software specialist with HP's San Diego Division, Frank Bockman has investigated a computer-aided work system and charting modules and has contributed to the development of measurement graphics software for the HP 7090A Measurement Plotting System. He was born in San Diego, California, came to HP as a student intern in 1980, and completed his BA degree in computer science from the University of California at San Diego in 1982. His professional interests include computer graphics, vector-to-raster conversion, and image rendering. Frank lives in Escondido, California, is married, and has a daughter. He is active in his church and enjoys soccer, racquetball, woodworking, gardening, and wine tasting.

32 Analog Channel

Jorge Sanchez



With HP since 1980, Jorge Sanchez attended San Diego State University, completing work for a BSEE degree in 1977 and an MSEE degree in 1981. His work on the HP 7090A Measurement Plotter includes designing the analog channel and the calibration algorithms for the analog channel as well as contributing to electromagnetic compatibility design. He is now developing new products as an R&D project manager. His previous professional experience was with National Semiconductor Corporation and with NCR Corporation. Jorge was born in Tecate, Mexico and currently lives in San Diego, California with his wife and two children. He is an avid sports fan and enjoys running, swimming, playing the piano, and gardening.

36 Usability Testing

Daniel B. Harrington

Dan Harrington holds degrees in physics from Albion College (BA 1950) and the University of Michigan (MS 1951). With HP since 1972, he has held a variety of management and engineering positions. He has been a product manager and an applications engineer and is now a publications engineer. He also worked at another company on the development and marketing of a mass spectrometer. He is named coinventor for a patent on a mass spectrometer and inventor for three other patents on various topics. Born in Detroit, Michigan, Dan lives in Corvallis, Oregon, is married, and has three children. He is president of the local Kiwanis Club and likes photography, music, camping, and travel.

A Stand-Alone Measurement Plotting System

This compact laboratory instrument serves as an X-Y recorder, a low-frequency waveform recorder, a digital plotter, or a data acquisition system.

by Thomas H. Daniels and John A. Fenoglio

MANY PHYSICAL PHENOMENA are characterized by parameters that are transient or slowly varying. If these changes can be recorded, they can be examined at leisure and stored for future reference or comparison. To accomplish this recording, a number of electromechanical instruments have been developed, among them the X-Y recorder. In this instrument, the displacement along the X-axis represents a parameter of interest or time, and the displacement along the Y-axis varies as a function of yet another parameter.

Such recorders can be found in many laboratories recording experimental data such as changes in temperature, variations in transducer output levels, and stress versus applied strain, to name just a few. However, the study of more complex phenomena and the use of computers for storage of data and control of measurement systems requires enhancement of the basic X-Y recorder. Meeting the need, Hewlett-Packard's next-generation laboratory recorder, the HP 7090A (Fig. 1), is a compact stand-alone instrument that can be used as a conventional X-Y recorder, a low-frequency waveform recorder, a digital plotter, and a complete data acquisition system.

X-Y Recorder Features

The HP 7090A Measurement Plotting System offers many improvements for the conventional X-Y recorder user. In the past, X-Y recorders have been limited to a frequency response of a few hertz by the response time of the mechanism. The HP 7090A uses analog-to-digital converters (ADCs) and digital buffers to extend the measurement bandwidth well beyond the limits of the mechanism. Each input channel has a 12-bit ADC capable of a 30-kHz sample rate. Since it is necessary to have about 10 samples/cycle for a good plot of the signal (remember, the minimum Nyquist rate of two samples/cycle only applies if there is a perfect low-pass output filter), this approach allows signals with bandwidths up to 3 kHz to be recorded.

The front-end amplifier presented many design chal-

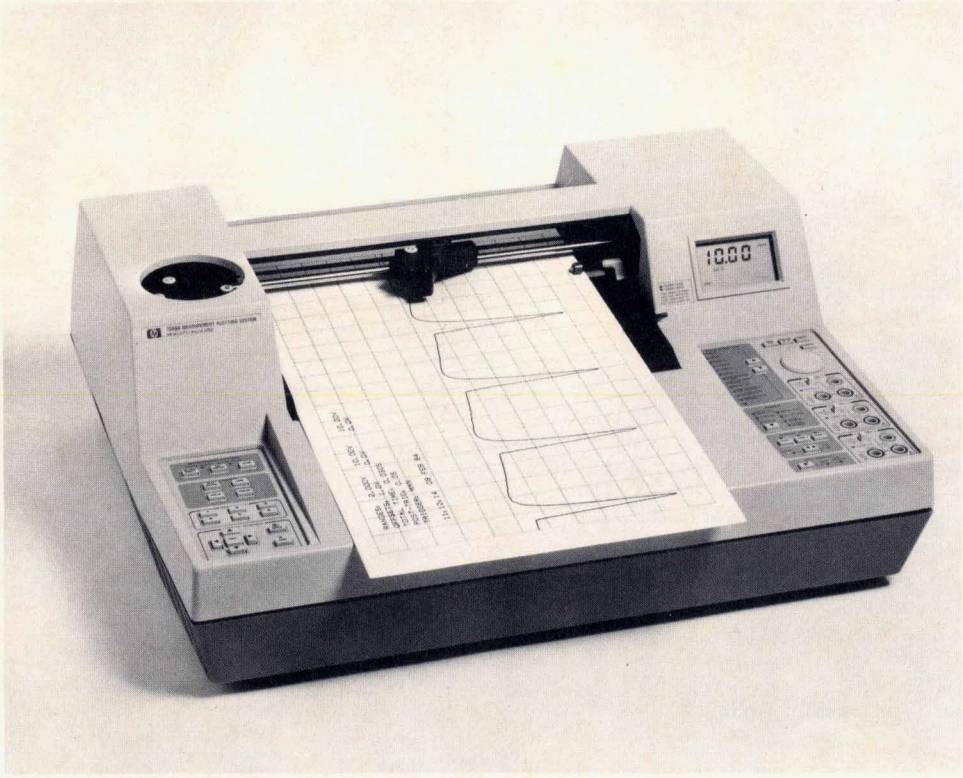


Fig. 1. *The HP 7090A Measurement Plotting System combines many of the features of an X-Y recorder, a low-frequency waveform recorder, a digital plotter, and a data acquisition system in one instrument that can be operated by itself or as part of a larger computer-controlled system.*

lenges. High common-mode rejection, high sensitivity, low noise, and static protection were a few of the more difficult areas. X-Y and stripchart recorders have used floating input circuitry to allow users maximum flexibility in connecting signals to the measuring device. The degree to which input signals can be isolated from chassis ground is specified as the common mode rejection (CMR). Achieving a high CMR means that the input circuitry *must not* be connected to chassis ground. This requirement posed a dilemma for a microprocessor-controlled system like the HP 7090A, because the microprocessing system *must* be connected to ground for noise prevention reasons. This design contradiction is resolved by using small independent power supplies for the front-end channels and by doing all of the data communication via optoisolator links. The point in the system where the floating circuitry is connected to the processing circuitry is shown by the optoisolator in the system block diagram (Fig. 2).

The most sensitive range of the HP 7090A is 5 mV full scale. The 12-bit resolution of the ADC allows measurements as low as 1 μV. Input amplifier noise and all external switching noises must be kept well below 1 μV over the full 3-kHz bandwidth. In addition, the standard HP design requirement of electrostatic discharge protection offered an even greater challenge—the same high-sensitivity floating input must be able to withstand 25-kV discharges directly to the input terminals! (See article on page 32 for details about the front-end design.)

The microprocessor is used for many functions, including signal processing on the raw analog-to-digital measurements. This makes it possible to calibrate the instrument digitally. Hence, there are no adjustment potentiometers in the HP 7090A (see box on page 22). During the factory calibration, a known voltage is applied to the input and the microprocessor reads the corresponding value at the output of the ADC. The calibration station then compares this value with the expected value. Any small deviation between the measured and expected values is converted to a calibration constant that is stored in the HP 7090A's nonvolatile memory (an electrically erasable, programmable read-only memory, or EEPROM). This constant is used by the internal microprocessor to convert raw measurement data to calibrated measurement data during the normal operation of the instrument. In addition, offset errors are continually checked and corrected during measurements. This helps eliminate the offset or baseline drifts normally associated with high-sensitivity measurements.

The use of a microprocessor also allows the user of an HP 7090A to select a very large number of calibrated input voltage ranges. Conventional approaches to input ranging usually involve mechanical attenuator switches with about fourteen fixed positions corresponding to fourteen fixed ranges. An uncalibrated vernier potentiometer is used for settings between the fixed ranges. The HP 7090A uses digitally programmable preamplifiers and attenuators. The gain of this circuitry can be set to 41,000 different values. The microprocessor commands different gain settings by writing to the front-end control circuitry via the optoisolator link.

Low-Frequency Waveform Recorder Features

The HP 7090A also can be used as a low-frequency waveform recorder. Triggering on selected input signal conditions allows a waveform recorder to capture transient events. In the HP 7090A, the triggering modes are expanded from the traditional level-and-slope triggering to include two modes of window triggering. The outside window mode allows for triggering on signals that break out of either an upper or a lower window boundary. The special inside window mode allows for triggering when the signal stays inside upper and lower window boundaries for the entire measurement period. The latter is the only effective way to trigger on a decaying periodic waveform like that caused by an ac power line failure (Fig. 3).

To implement the sophisticated triggering capability described above, the HP 7090A uses digital triggering techniques. No analog circuitry is involved. The trigger decision is made by looking at the digitized input data that comes from the ADCs and comparing this to the desired trigger conditions set by the user. At the higher sampling rates the microprocessor is not fast enough to make trigger decisions unaided. Therefore, a semicustom LSI circuit is used to augment the processor in this area. This IC is a CMOS 770-gate array especially programmed to do input data buffer management. It is shown in the system block diagram as the front-end gate array.

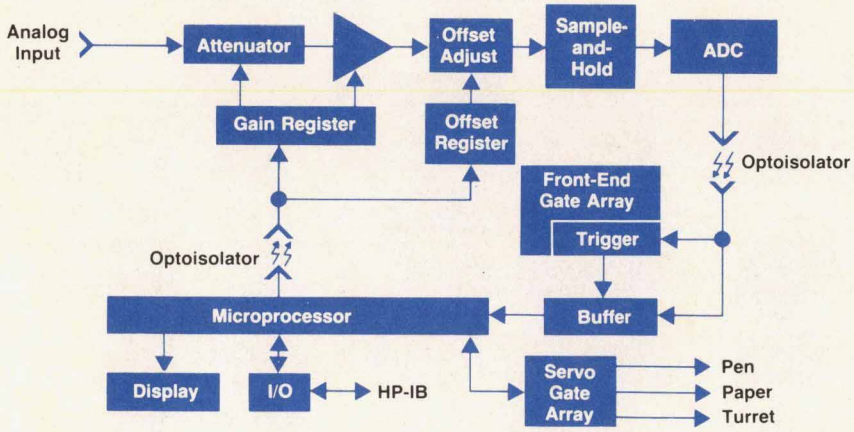


Fig. 2. *Simplified block diagram of the HP 7090A.*

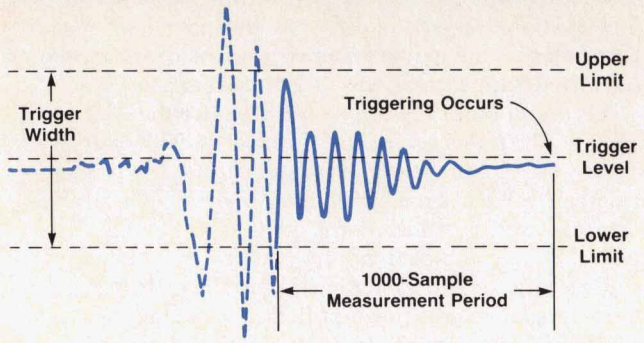


Fig. 3. *The HP 7090A's special inside window triggering mode allows capture of waveforms that remain inside the window for the measurement period. In the above example, the trigger occurs after the thousandth consecutive sample that is inside the window defined by setting the* ***TRIGGER LEVEL*** *and* ***TRIGGER WIDTH*** *controls on the front panel. This enables the recording of such events as a decaying periodic waveform caused by an ac line failure.*

One final measurement feature is the highly accurate, wide-dynamic-range, time-axis capability that comes about because the HP 7090A's time axis is computed by dividing the system's crystal-controlled clock frequency. This allows for time sweeps from 0.03 second to 24 hours full scale.

Recorder/Plotter Features

The desire to produce a single product that could be used as a continuous X-Y recorder and as a full-performance digital plotter created many different performance objectives for the sophisticated servo system found in the HP 7090A. It uses three separate servo algorithms, each optimized for a specific task. In the digital plotter mode, the servo must match both axes and faithfully draw straight line vectors between endpoints.

Plotting data from the digitized input signal buffers also requires the servo to draw vectors between the data points, but there is a subtle difference. In this case, the servo can be optimized to look ahead at the next few data points and

Eliminating Potentiometers

Potentiometers are not needed in the HP 7090A Measurement Plotting System because its internal microprocessor:

- Controls the front end
- Determines the gain constants
- Performs the offset calibration
- Corrects the data.

The microprocessor has the ability to write to three main ports in the front channel (see Fig. 1). The first port controls the FET switches and relays that govern the coarse gain settings and the relay that passes the input signal into the front-end amplifiers. The second port determines the amount of offset fed into the input signal. The third port establishes the attenuation that the signal sees by means of the digitally programmable amplifier. This port governs the fine gain settings.

There are 14 coarse gain settings covering a span of 5 mV to 100V, inclusive. While an HP 7090A is on the assembly line, it passes through a calibration of the gain of each of the three channels at each of the coarse gain settings. This calibration procedure produces a two-byte number for each channel at each setting, and then stores these numbers in nonvolatile memory. To determine these numbers, an offset calibration is performed (as discussed later) and a voltage equal to the full-scale voltage is placed on the inputs of the channel. For example, if the full-scale voltage of the front end is set to 200 mV, a 200-mV dc signal is placed on the inputs. The buffer is filled using a 250-ms/measurement time base and 200 of the uncorrected analog-to-digital samples are sent over the HP 7090A's HP-IB (IEEE 488) to the controller, an HP Series 200 Computer. These samples are not internally corrected; they are the direct output of the ADC in the instrument's front end. These samples are averaged, and the average A is put into the following formula:

$$\text{Gain constant} = \left(\frac{1974}{A - S - 2048}\right) \times \left(\frac{\text{DVM}}{\text{Ideal Volts}}\right)$$

where DVM is the voltage read by a digital voltmeter of the input voltage to the front end, and Ideal Volts corresponds to the full-scale voltage that should be on the input. S is the software offset found by the offset calibration. The typical result is about 1.03. The word stored in the nonvolatile memory is the gain constant

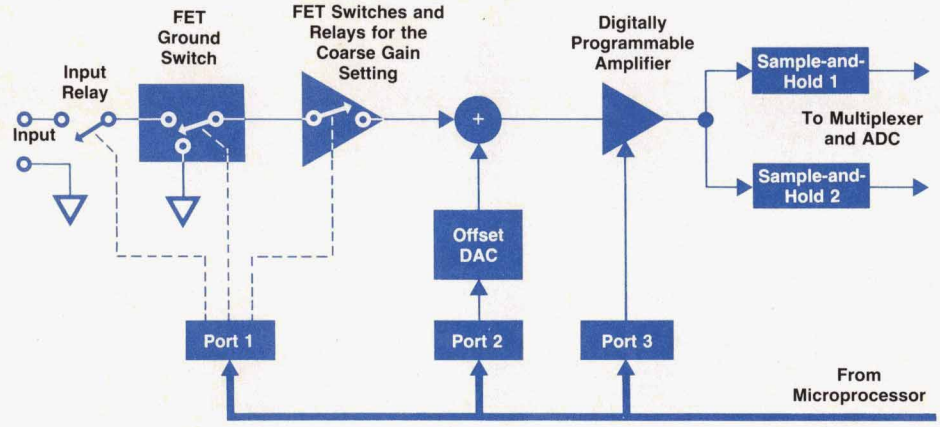


Fig. 1. *Block diagram of front-end section of the HP 7090A.*

minus one. The above procedure for finding the gain constants is repeated for each of the 14 ranges, for each of the HP 7090A's three channels.

The number 1974 in the above equation comes from the full-scale input to the ADC being mapped to 4022 − 2048 + 1974, rather than 4095, so that some margin can exist at the upper and lower limits of the analog-to-digital output. This allows for some offset variation in the front-end electronics.

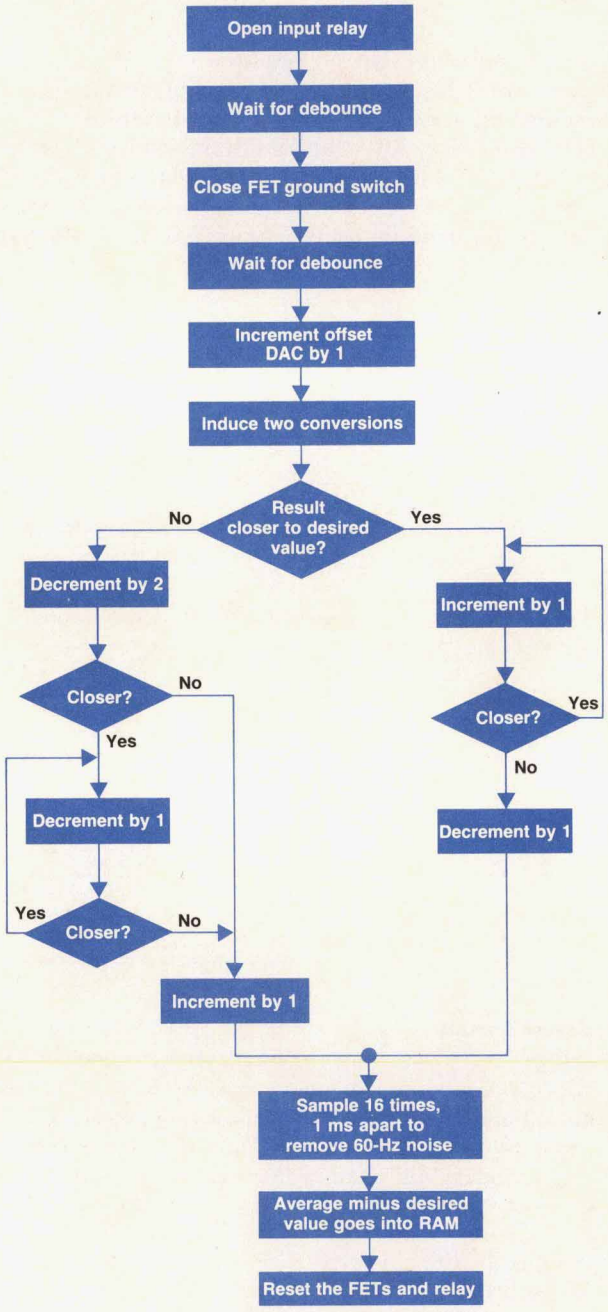


Fig. 2. *Flow chart of front-end calibration procedure for each channel of the HP 7090A.*

In-Service Autocalibration

At any point in time, there is some offset in the front end. This offset can change because of such factors as temperature and the age of the components. Therefore, there is a need to calibrate the instrument while it is in operation, and even during a measurement. The internal real-time clock is useful in telling the HP 7090A's internal processor when to perform an internal calibration. Generally, such a calibration is done every 7 to 10 minutes.

The procedure (Fig. 2) followed for correcting the offsets in one channel begins with opening the input relay—the one that allows the input signal to pass through the front end. Next, a FET is turned on, which grounds the input to the amplifiers. There are appropriate delays to let the relay and FET debounce and settle to fixed values. The processor is then able to induce the ADC to convert the zero input twice. The two samples come from the two sample-and-hold sections within the front end. The resulting values are stored in RAM. Next, the offset port is written with a number equal to one plus the original value. The processor induces two more conversions, and the new values are compared with the previous values stored in RAM. If the new pair of values is closer to the desired zero value, based on internal computations of the range and offset settings, the offset port value is incremented again and the process of comparison is repeated. If the new values are farther than the previous set from the desired value, then the offset port value is decremented twice, and two new values are found and compared with those for the original offset port number. If the new values are closer to the desired value, the offset port value is decremented once and the process is repeated. The process stops when the most recent values from the ADC are farther than the previous values from the desired value.

The processor reverses the trend of incrementing or decrementing the offset port value once leaving the offset DAC at its optimal value, takes 16 samples one millisecond apart for each sample-and-hold, and averages these samples to eliminate any 60-Hz noise. The two averages have the desired offset value subtracted from them, and the two differences are stored in RAM. The result is that the offset port is at its optimal value and two 16-bit words are stored that correspond to the residual offsets of the front end and each sample-and-hold. These words are called the software offsets, and are used in correcting the data. The zero FET is turned off and the input relay is closed. The front end is now calibrated and ready for sampling the external input.

When the ADC samples data, its output must be corrected for gain and offset. Each time a conversion takes place, a 10-bit counter is incremented and the least significant bit is the index for which sample-and-hold (1 or 2) corresponds to the data sample. The uncorrected data is inserted into the following formula:

$$(D_i - V_{osi} - \text{Ideal Zero}) \times GF(J) + \text{Ideal Zero} = D_{corrected}$$

where D_i corresponds to the uncorrected data of sample-and-hold i (i = 1 or 2), V_{osi} equals the software offset for sample-and-hold i, Ideal Zero is the binary equivalent of the offset scaled to 0 to 4095 where 2048 represents a zero offset, and GF(J) is the gain factor word stored in the EEPROM plus a word for range J (J = 1 through 14, corresponding to the 5-mV through 100V ranges).

Stephen D. Goodman
Development Engineer
San Diego Division

adjust its acceleration profile to reduce the plot time by removing the need to come to a complete stop after each data point. When in the **RECORD DIRECT** mode, the digitized input signal data is fed directly to the servo control system, bypassing the data buffers, and the pen follows the input signal in the continuous (nonvector) manner of conventional X-Y recorders.

The servo system uses the familiar dc motors and optical position encoders that are common to all modern digital plotters. But unlike such plotters, this servo system uses an algorithm that closes the servo loop and allows the device to emulate the analog-like characteristics of traditional X-Y recorders. This is done by using the microprocessing system and another semicustom LSI circuit, a CMOS 2000-gate array. This hardware combination allows the processing system to model the characteristic block diagram of a traditional analog servo system in a manner fast enough to appear real-time to the user when recording slow-moving signals (under a few cycles per second). In this mode, the HP 7090A performs in exactly the same manner as a conventional X-Y recorder.

Another feature of the HP 7090A is its ability to draw its own grids. No longer is the user forced to try to align the desired measurement to a standard inch or metric grid. The user simply specifies the required number of grid divisions, from one to one hundred, by using the HP 7090A's front-panel controls. A firmware algorithm is invoked by pressing the front-panel **GRID** button, which then draws the specified grid between the specified zero and full-scale points.

The graphs created by the HP 7090A can be used for observing the trends of the measurement. The high-accuracy measurement made possible by the 12-bit ADC can be appreciated further by using the internal character generator to annotate any desired data point with three-digit resolution.

The processor also makes possible other features that enhance the measurement display capability of the HP 7090A. A calendar clock IC backed up with a battery and connected to the processor can be used to provide labeling of time and date at the push of a front-panel button. A nonvolatile memory (EEPROM) IC stores front-panel setup conditions, and two internal digital-to-analog converters convert digital data in the buffer memory to analog signals that can be displayed on a conventional oscilloscope to preview the buffer data, if desired, before plotting.

Data Acquisition System Features

The HP 7090A can be used as a computer-interfaced data acquisition system by using its built-in HP-IB (IEEE 488) I/O capabilities. All setup conditions and measurements can be controlled remotely by using an extension of the HP-GL (Hewlett-Packard Graphics Language) commands tailored for measurements. The data in the buffer can be transferred to a computer. The computer can process the data and then address the HP 7090A as a plotter to display the results.

The HP 17090A Measurement Graphics Software package (see article on page 27) was developed to provide user-friendly access to the many measurement capabilities of the HP 7090A.

Acknowledgments

The final design challenge was to offer the above capabilities without increasing the price above that of a conventional X-Y recorder. We would like to thank the many departments of HP's San Diego Division that helped make this dream a reality.

Digital Control of Measurement Graphics

by Steven T. Van Voorhis

THE OBJECTIVE of the servo design team for the HP 7090A Measurement Plotting System was to develop a low-cost servo capable of producing quality hard-copy graphics output, both in real-time directly from the analog inputs and while plotting vectors either from the instrument's internal data buffer or received over the HP-IB (IEEE 488) interface. The mechanical requirements of the design were met by adopting the mechanics of the earlier HP 7475A Plotter. This approach had the significant advantage of a lower-cost solution than could have been achieved with a new design. What remained then was to design the electronics and firmware for reference generation and control of the plant (dc servo motor and mechanical load).

Servo Design

Fig. 1 is a block diagram of the major components of the HP 7090A servo design for one axis, there being no significant difference between the pen and paper axes for the purposes of this discussion. Fig. 2 shows the corresponding servo model. The plant is modeled as a system with the transfer function of $K_m/(s+P_e)(s+P_m)$. Feedback of position and velocity was found to give sufficient control to meet the line-quality objectives.

The prime mover for each axis is a low-cost dc servo motor. Feedback of motor shaft position is provided by a 500-line optical encoder. By detecting all state changes of the two-channel quadrature output of the encoder, 2000 encoder counts per revolution of the motor shaft can be detected. This yields an encoder resolution of slightly bet-

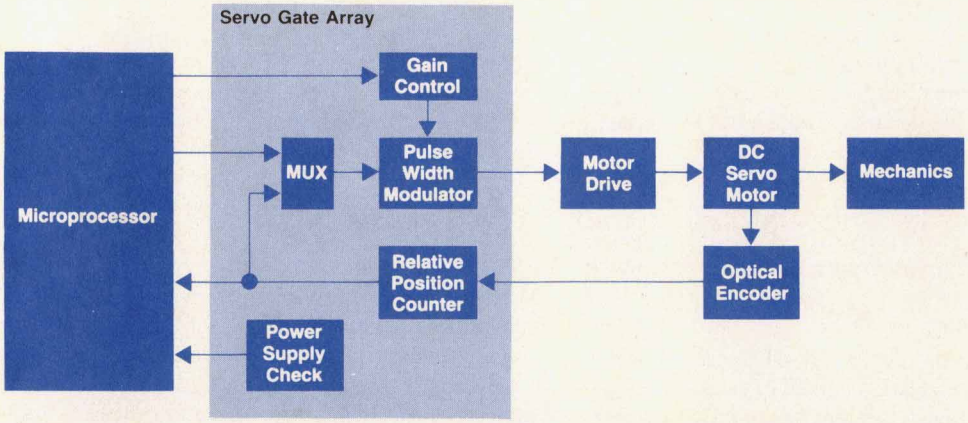


Fig. 1. *Block diagram of HP 7090A servo system.*

ter than 0.001 inch at the pen tip. Since the feedback is derived from the motor shaft and not the pen tip, any plant dynamics between these two points are open-loop with respect to the servo system. It is therefore essential that the mechanics be "stiff" between the motor shaft and the pen tip within the 100-Hz bandwidth of the servo system.

The digital electronics portion of the control loop is implemented in a single gate array of some 2000 gates packaged in a 40-pin dual in-line package. The two-channel quadrature feedback signals from the optical encoders are decoded within the gate array to clock two 8-bit relative position counters, one for each axis. The position counters are cleared on each read by the microprocessor, in essence providing velocity feedback to the microprocessor. The microprocessor integrates this feedback to generate position information. The power supply check circuitry provides the microprocessor with a 6-bit measurement of the motor drive supply voltage.

In the feed-forward path, the microprocessor controls each motor by writing to two 8-bit registers for each axis in the gate array. The two registers control the period and duty cycle of the pulse-width-modulated motor drive signals. Pulse-width-modulated motor drive circuits were chosen because of the ease of interfacing to digital systems and their efficiency advantage over linear drivers. Using the feedback of the motor drive supply voltage, the microprocessor can adjust the period of the drive signal to regulate the gain of the drive path. This eliminates the expense of having a regulated supply for the motor drivers. The microprocessor varies the duty cycle of the pulse width modulator as dictated by the solution of the control equations to achieve control of the plant.

When sampling the front-end channel at high sample rates, there is not sufficient processing power available from the 6809 microprocessor to execute both the channel and the servo routines in real time. Thus, a multiplexer under microprocessor control is provided to allow the gate array to close a position loop about the plant without microprocessor intervention. To avoid any instability caused by loss of velocity information, the position loop gain is halved when this is done. This allows the microprocessor to supervise the channel data transfer without the overhead of executing the servo routines. Other miscellaneous circuitry in the servo gate array provides pen-lift control, the microprocessor watchdog timer, the front-end channel communications serializer, and a chip test.

The real-time servo routines are initiated by a nonmaskable interrupt, which is run at a 1-kHz rate while plotting. Aside from various housekeeping duties, the main responsibilities of the servo routine are to maintain control of the plant by closing the feedback loop, and to generate the reference inputs to drive the system.

Closing the feedback loop is always done in the same manner while plotting either vectors or data directly from the front-end channels. The relative position register is read and summed with the old plant position to generate the updated plant position. A copy of the relative position register value is multiplied by the velocity feedback constant to generate the velocity feedback term. The plant position is subtracted from the reference input to generate the position error. From this, the velocity feedback term is subtracted and a deadband compensation term is added to generate the control value to be sent to the pulse width modulator. The power supply check register is read and the period of the pulse width modulator is adjusted to ensure a constant gain for the motor drive block.

Plotting Data

There are three separate reference generators that can be invoked, depending on the mode of plotting. The first is for direct recording of the front-end channel data, the second is used when plotting vectors parsed from the I/O bus (HP-IB), and the third is used when plotting from the HP 7090A's internal data buffer. When directly recording front-end channel data, the inputs are continuously sampled at 250 Hz and the internally generated time base is updated at the same rate. The samples are scaled according to the

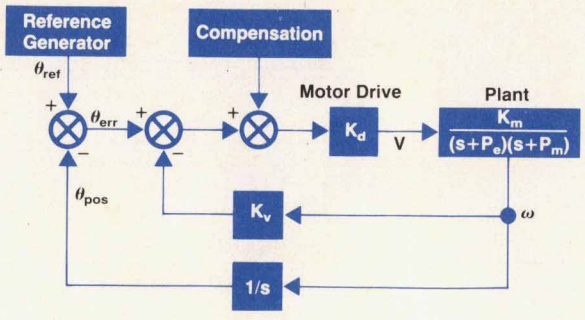


Fig. 2. *Model of servo in Fig. 1.*

prevailing setup conditions to provide the new desired position for the pen and paper axes. Were these inputs fed directly to the servos, high-frequency or noisy input signals could easily result in motor overheating. The new desired positions are therefore passed to the servos through a reference profiler, which limits plant acceleration to 2g and maximum slewing speed to 50 inches per second. This limits the power input to the motors to a safe operating level and preserves acceptable writing quality. This approach results in no overshoot when recording step inputs and provides good reproduction of 1-cm peak-to-peak sinusoidal waves for frequencies below 10 Hz.

When the HP 7090A operates as a plotter, HP-GL* commands received over its HP-IB interface are parsed in accordance with the current graphics environment to generate new desired pen and paper locations. These new locations are represented as two-dimensional vectors relative to the present location. These vectors are passed to a vector reference generator via a circular queue capable of storing up to 30 vectors. The vector reference generator takes vectors from the queue and profiles the input to the servos to constrain the plant to a constant 2g acceleration and 75-cm/s maximum vector velocity. Fig. 3 depicts the profiling of two consecutive vectors. The second vector is long enough for the pen to reach maximum velocity and then to slew at this velocity for some time before the onset of deceleration.

A short pause of 12 milliseconds between vectors ensures settling of the plant at the vector endpoints. The references for the paper and pen axes are simply scaled from the vector profile by the cosine and sine, respectively, of the angle between the vector and the positive paper axis.

Vector Profiler

Plotting from the internal data buffer could be performed in exactly the same manner as plotting vectors from the HP-IB interface. However, several attributes of this mode of plotting led to the development of a new reference generator. The first is that for each channel to be plotted, a string of 1000 vectors is already stored in the internal data buffer. Thus, the overhead of running the HP-IB interrupt routines, the parser, character generator, and other routines to create vectors is eliminated. Second, since the functions to be plotted are continuous, the 1000 data points form a contiguous string of short vectors (typically less than 0.025 inch), all plotted pen down. Furthermore, the angle formed between any two consecutive vectors is typically very shallow.

*Hewlett-Packard Graphics Language

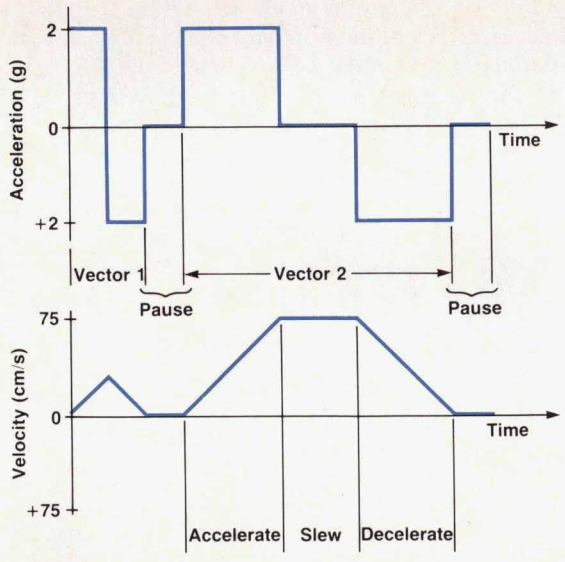


Fig. 3. *Profiling of two typical vectors parsed from the HP-IB.*

Consider the trivial case of plotting a dc signal from the internal data buffer. Assuming a 15-inch trace on B-size paper, this amounts to plotting 1000 vectors, each of length 0.015 inch, all along a straight line. Using the HP-IB vector reference generator would require 10 ms to profile the acceleration and deceleration of each vector, plus a 12-ms intervector delay. Thus, it would require 22 seconds to draw this 15-inch line, whereas if it were plotted as a single vector at 75 cm/s, it would require just over 0.5 second. Therefore, a new vector profiler was designed for plotting from the internal data buffer with the objective of improving throughput. This algorithm does not require a stop at each vector endpoint. Rather, it constrains the vector endpoint velocity so that the following three conditions are met:

- The angle drawn at the vector endpoint is drawn with negligible error.
- The vector is not drawn in less than eight iterations of the servo interrupt routines (i.e., 8 ms).
- A 2g deceleration to a full stop at the end of the vector string is achievable.

Using this internal data buffer reference profiler, a 15-inch dc signal trace is plotted in 8 seconds, because of the second constraint. This is nearly a factor of three in throughput improvement compared to using the HP-IB vector reference generator. In fact, many functions are plottable in the 8-second minimum time with this technique, resulting in throughput gains as high as eight.

Why not apply the same profiling technique to vectors received over the HP-IB interface? The answer is twofold. First, vectors plotted from the bus are generally not contiguous strings representing continuous functions. They typically have many pen up/down cycles, form acute angles, and are longer, all of which reduce the throughput gain using this algorithm. Second, applying the three conditions to determine the vector endpoint velocity requires additional processing of each vector to check angles and determine the distance to the end of the current string of vectors. To do this in real time requires that, as each new vector is received, the processor backtrack through the list of current unplotted vectors to see if their endpoint velocities can be increased. When the nature of the plot is such that little throughput gain is possible from the application of these algorithms, the additional processing load of executing them can actually result in a throughput loss. Therefore, this approach is restricted to plotting of the internal data buffers where the throughput gains are the greatest.

Measurement Graphics Software

by Francis E. Bockman and Emil Maghakian

HP 17090A MGS IS A SOFTWARE PACKAGE written for the HP 7090A Measurement Plotting System that runs on HP's Series 200 Computers. MGS allows the user to:

- Set up measurements
- Take measurements
- Store and retrieve measurement data to and from disc files
- Annotate measurements with text, axes, and simple graphics
- Manipulate measured data
- Provide soft and hard copy of measured and manipulated data.

MGS was written to provide a system solution to some of the general problems of measurement recording and data acquisition. It is designed to be used by scientists and engineers not wanting to write their own software. This software package extends the capabilities of the stand-alone HP 7090A.

The package consists of two discs. The first disc contains the core of the system, the initialization routines, the library routines, and the memory manager. The second disc contains six code modules, one for each functional subsystem. The measurement setup module contains code to help the user specify the setup parameters relevant to the measurement. The measurement module allows one to start the measurement and the flow of measurement data into the computer. The storage-retrieval module contains code to store and retrieve measurement data and setup information to and from disc memory. The data manipulation module implements the ability to manipulate measurement data mathematically. The annotation module adds the capability of adding graphical documentation to the measurement record. The display module allows a user to display the measurement data taken and the annotation on either the computer's display screen or on paper.

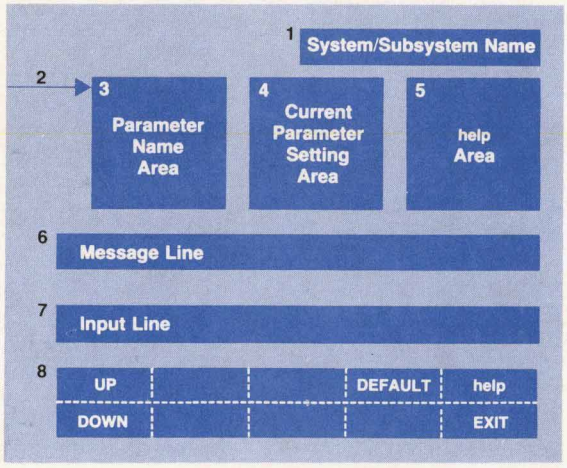


Fig. 1. *Screen layout for MGS.*

Since MGS is intended for the instrument/scientific market where users typically write their own instrument control software, we used BASIC as the application language. Hence, users of the package can add their own code in a commonly understood language to tailor it to their specific needs. The application is distributed in source form.

Human Interface

The human interface is designed for both novice and expert users. We have made the assumption that all our users are familiar with X-Y recording, and that they have used recorders for data measurement.

A human interface should be self explanatory and descriptive to accommodate a novice user. An expert user, on the other hand, requires an interface that is like a tool—one that does not hamper creativity and does not ask a lot of questions (conversational).

MGS's human interface is an extension of the HP 7090A's human interface. There are no operational conflicts between the HP 7090A and MGS.

Screen layout is an important part of every human interface. We have made a special effort to ensure a consistent screen layout (Fig. 1) throughout the modules to improve the feedback to the user. Fig. 2 is an example of an actual CRT display for MGS. The definitions for the various elements of the screen layout are:

1) Subsystem Name. This is the name of the subsystem. Each box on the design tree (Fig. 3) is a subsystem. For instance, the DISPLAY functional area is composed of the CHANGE SETUP, SCREEN, and PLOTTER subsystems. The CHANGE SETUP subsystem also has under it the CHANGE SCALE subsystem (not shown).

2) Arrow. The user can only change one parameter setting at a time. The arrow points to the parameter that is currently modifiable. The arrow is controlled by the softkeys UP (**k0**) and DOWN (**k5**).

3) Parameter Name Area. This area of the CRT is where the parameter names are displayed.

4) Current Parameter Setting Area. The current parameter

```
                                        MEASUREMENT SETUP
--> Measurement          <System>
    Units

    Recorder Mode
      Dependent vs.      <Ch 1>
      Independent        <Time>

    Measurement          <Buffered>
    Mode

    Trigger
      Mode               <Manual>
      Post/Pre           <0.00>

    Total Time           <1.00 Sec.>
```

Fig. 2. *MGS control display.*

setting is displayed on the same line as the parameter name. The parameter setting is enclosed by angle brackets. For example:

```
param1                  <param1 setting>
--->param2              <param2 setting>
  .                       .
  .                       .
param N                 <param N setting>
```

where parameter 2 is currently selected to be modified.

5) Help Area. This area of the CRT is used to display help information to the user, which will consist of either the current valid range of parameter settings for the parameter designated by the arrow, or information on how to set the parameter, or what to set the parameter to.

6) Message Line. This line is used by the software to display messages to the user. When applicable, it will specify the permissible range of parameter settings.

7) Input Line. This line is used for entering text and numbers when required by MGS.

8) CRT Softkey Labels. This area displays the labels for the HP 9000 Series 200 Computer's softkeys. The labels shown in Fig. 1 do the following actions when the corresponding softkeys are pressed:

UP (**k0**) : Places the arrow up one parameter.

DOWN (**k5**) : Places the arrow down one parameter.

DEFAULT (**k3**) : Sets the current menu parameters to their default settings.

help (**k4**) : This softkey has an on/off toggle action An asterisk in the softkey label implies the help information will be displayed in the help area on the CRT, for the current menu and all the following menus. This softkey may be toggled on and off as many times as necessary.

EXIT (**k9**) : Returns the user up one level of the tree to the previous subsystem.

The primary user input to the software is the knob and the softkeys on the keyboard of the Series 200 Computer. Input from the keyboard has been limited as much as possible. The softkeys provide the user with the ability to control the flow through the design tree (Fig. 3).

The knob controls the setting of the parameter selected by the arrow on the menu. To set any parameter, the knob must be rotated first. The software will then react in one of the ways listed in Table I.

Table I

Parameter Type	Software Reaction
Enumerated (i.e., specific list of settings)	Turning the knob will scroll through the current valid parameter settings for the specified parameter.
Positional	Turning the knob will move the graphics cursor in a left or right direction. Turning the knob with the **SHIFT** key held down will move the graphics cursor in an up or down direction.
Number with limited range	Turning the knob will cause the parameter setting to be incremented or decremented by a small amount. Turning the knob with the **SHIFT** key held down will cause the parameter setting to be incremented or decremented by a large amount.
Text or number with unlimited range	Turning the knob will cause a message to be displayed on the message line and the current setting to be displayed on the input line. Then the user may modify this setting by typing in the new setting and pressing the **ENTER** key when correct.

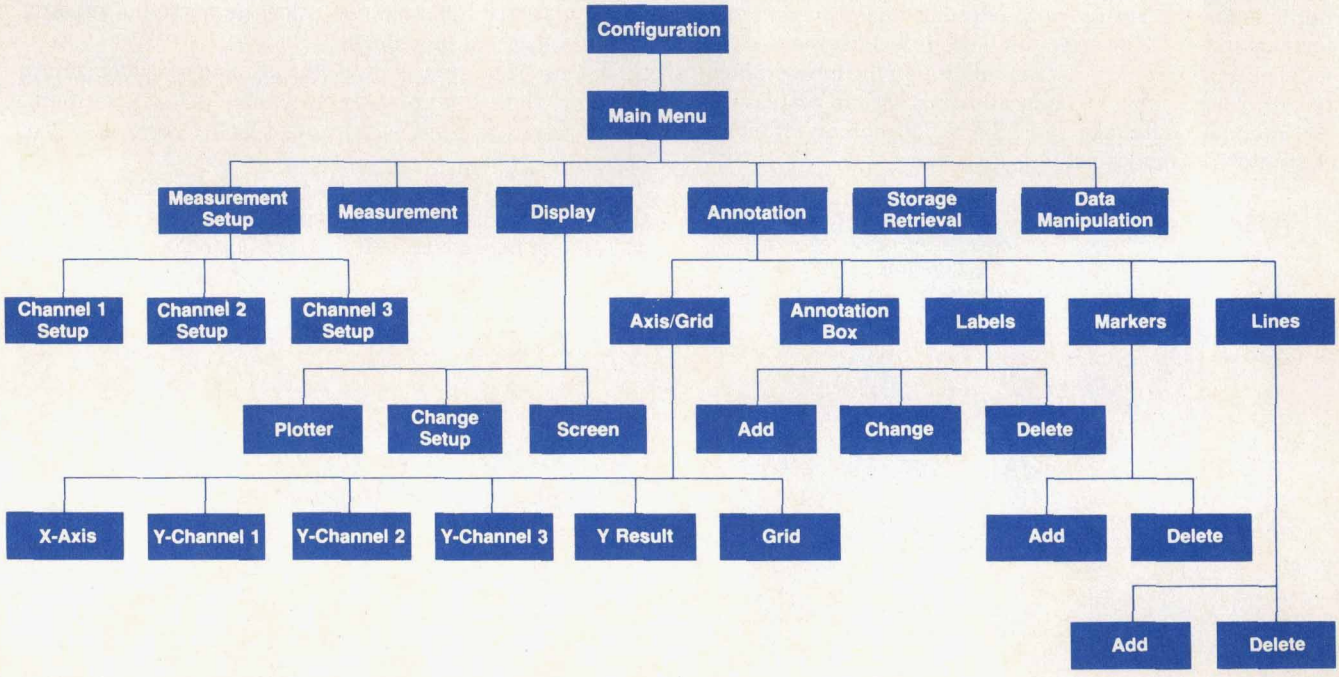


Fig. 3. *Conceptual layout of MGS.*

The major philosophy in this human interface is "modification, not specification." This means that at all times the system settings are valid, and user can change one valid setting to another. The user is not burdened by descriptions or questions. The help area describes the current possible settings. It is placed to one side intentionally so it does not interfere with the man-machine interface. It can be turned on and off at the user's discretion.

The design of the human interface limits the number of error states. The user can only see an error message when entering a number from the keyboard or using the HP 7090A to enter a voltage. We have managed to achieve this goal by updating the next possible state lists every time a parameter is modified.

Overall Design

There is a menu associated with every mode of the design tree (Fig. 3). The tree is three levels deep from the main level. The main level consists of the menu that allows access to the six major functional modules: measurement setup, measurement, display, annotation, storage/retrieval, and data manipulation. The softkeys are labeled according to their function; pressing a softkey will place the appropriate menu on the CRT. The general rule is that a user exits a menu to the same menu(s) the user went through to enter the menu. Pressing the EXIT softkey returns the user up one level of the tree. The configuration level is a one-time process and is only entered at the start of the program. Pressing the EXIT softkey at the main level will stop the program after verifying that the user really wants to exit.

Core Library and Swapper

The software package consists of a core or kernel that must always reside in memory. There is additional code for initialization and configuration that is loaded initially and then removed from memory after running. The six main code modules that implement the functionality of the system can be either resident in memory or loaded from disc, depending on the system configuration and available memory. There is also a library of utility routines that resides in memory with the kernel. The library contains code to handle the screen menus and data structures. Also, the code that communicates with the HP 7090A for data transmission resides in the library.

A part of the system known as the swapper, or memory manager, is responsible for ensuring that there is enough memory available for requested operations. At program initialization time, the swapper loads in the whole system if there is enough memory; if not, it loads just the main section of the system and the supporting libraries. Provided enough memory exists for the former action to take place, the swapper will not need to take further action. Assuming there is insufficient memory to load the complete system, the swapper will take actions when memory allocation is needed. The swapper handles all requests to enter a subsystem from the main menu. It first checks to see if the subsystem is in memory. If it is, no action is taken by the swapper and the subsystem is entered. If the subsystem is not in memory, the swapper checks to see if enough memory is available to load it in. If so, it is loaded and entered. Otherwise, space in memory will be made available by removing other subsystems not needed.

Data Structures for Menus

As mentioned earlier, all the menus in MGS are consistent. There is a single data structure that contains all the data for a screen. The diagram in Fig. 4 gives a graphical representation of the logical structure and Table II defines the elements shown in Fig. 4.

MGS prevents the user from entering error states. This task is done by changing o_strt and o_cnt entries for a given attribute. All the valid entries for attribute p are always between o_strt(p) and o_strt(p)+o_cnt(p).

This data structure is built using one and two-dimensional arrays in BASIC. There are several copies of this structure, one for each screen layout. The data definition portion of MGS would have been much smaller and storage more efficient if BASIC had dynamic storage allocation capability like Pascal.

Data Structure for the Knob

MGS relies heavily on the knob of the Series 200 Computers for input. At times the knob is used for entering a numeric value, such as volts at full scale, total time, etc. To make the knob more useful we had to make it nonlinear.

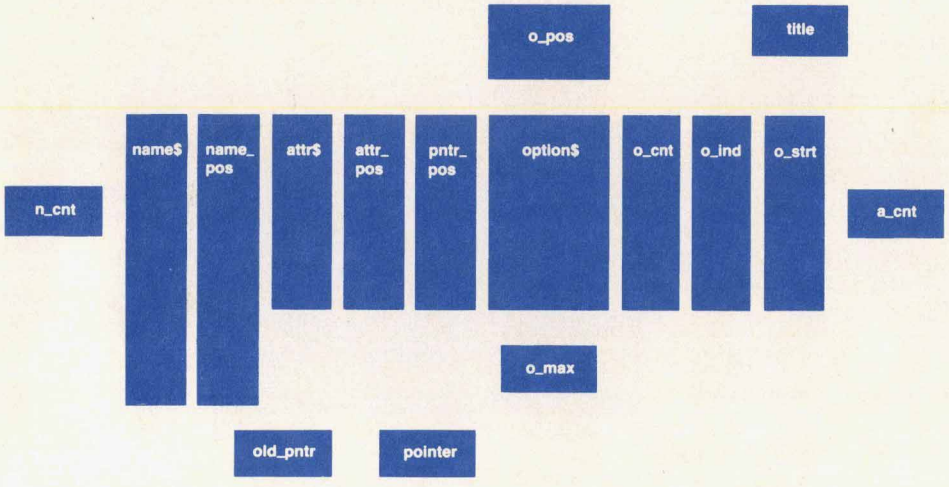


Fig. 4. *Graphical representation of MGS data structure for a screen display.*

Table II

Element	Definition
name$	Holds parameter names
name–pos	Holds the encoded x-y position of the names on the screen. There is one entry for every name$ entry. Instead of using two integer arrays for keeping the x and y positions, the following encoding scheme is used: name–pos(i)=x(i)*512+y(i). This is done to conserve storage space.
n–cnt	Holds the number of entries in name$ and name–pos columns.
attr$	Holds the current parameter setting.
attr–pos	Holds the x-y position of where parameters are to be displayed.
pntr–pos	Holds the x-y position of where the pointer is to be displayed for each parameter.
option$	A two-dimensional structure. Each row of this structure holds all the possible settings for the corresponding parameter.
o–cnt	Holds the count of valid entries per row in the option table.
o–ind	Holds the current index of the item from the option table that is being displayed, that is, attr$(i)=option$(i,o–ind(i)).
o–strt	Holds the logical first entry in option$.
o–pos	Holds the x-y position of where options are to be displayed.
o–max	Maximum number of options in the option table.
title	Holds a string that contains a screen name.
pointer	Points to the current row in the option table. The UP and DOWN softkeys change the value of this variable.
a–cnt	Holds the number of parameters in the data structure.
old–pntr	Holds last value of the pointer.

This means the step size of the knob is dependent on the current value of the knob. For example, when the current value for volts at full scale is between 0 and 1V, the increment is 0.05V, and when the current value is between 50 and 100V, the increment is 1V.

To make this task uniform throughout MGS the data structure outlined in Fig. 5 is used.

Each table contains several rows of data. Each row is for a given range. Table III defines the parameters.

Table III

Element	Definition
increment	Holds the value by which the current setting will be incremented.
lower_bound	Holds the minimum limit of the range.
upper_bound	Holds the maximum limit of the range.
c_low_ind	Holds the first legal row of the table.
c_hi_ind	Holds the last legal row of the table.
c_index	Points to the current row in the table.
c_curr	Holds the current value. This is the variable that is being incremented and decremented.

c_low_ind and c_hi_ind are used to control the legal limits of the knob. Valid limits are kept between the high and low indexes.

The following conditions are used for moving up and down in the table:

If c_curr > upper_bound(c_index) then c_index = c_index+1 and c_curr = lower_bound (c_index)

If c_curr < lower_bound(c_index) then c_index = c_index−1 and c_curr = upper_bound (c_index)

Every time the value of c_index is changed, the following condition must be checked:

If c_index > c_hi_ind then c_index = c_low_ind

If c_index < c_low_ind then c_index = c_hi_ind

There is a copy of this data structure for every numeric parameter. Again, this is because of the limitations of BASIC.

Measurement Setup Module

In this module, the user sets up an experiment and specifies dependent channels, independent channels, triggering mode, duration of experiment, type of experiment, etc. Accessible through this module are channel setup modules. In those modules the user sets range, offset, and trigger level and width for each channel. If the measurement is to be conducted in user units, the user specifies the minimum and maximum user units, instead of range and offset.

Up to now, most users of X-Y recorders had to convert their units to voltage levels, and then take a measurement in volts. Finally, they had to convert volts back to their units. This is also the case with the stand-alone HP 7090A.

MGS allows the user to set up an experiment in volts. This is provided for the sake of consistency with the stand-alone machine. In addition to volts, MGS gives the user the capability of setting up and taking a measurement in some other unit system: displacement, acceleration, force, saturation, etc. To set up a measurement in volts, the user specifies range and offset settings for each channel and trigger information for Channel 1, just as for the stand-alone HP 7090A.

When in user units, a measurement is set up by specifying the minimum and maximum possible readings in user units for each channel and trigger information for Channel 1. Trigger information is specified in user units. We believe that the availability of user units enhances the usefulness of MGS. For example, in measuring temperature in a chemical experiment, we can set user units limits for Channel 1 to −100°C and 100°C and set the trigger level to 10°C.

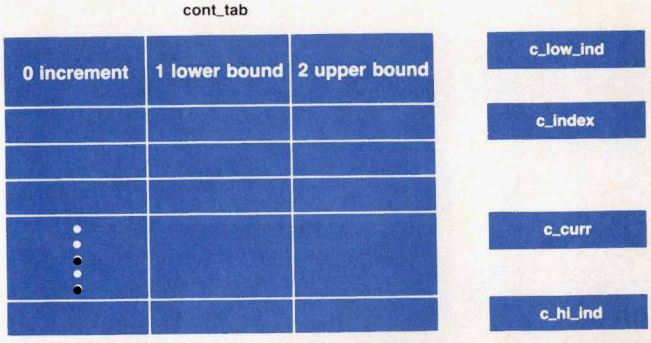


Fig. 5. *Data structure for knob control.*

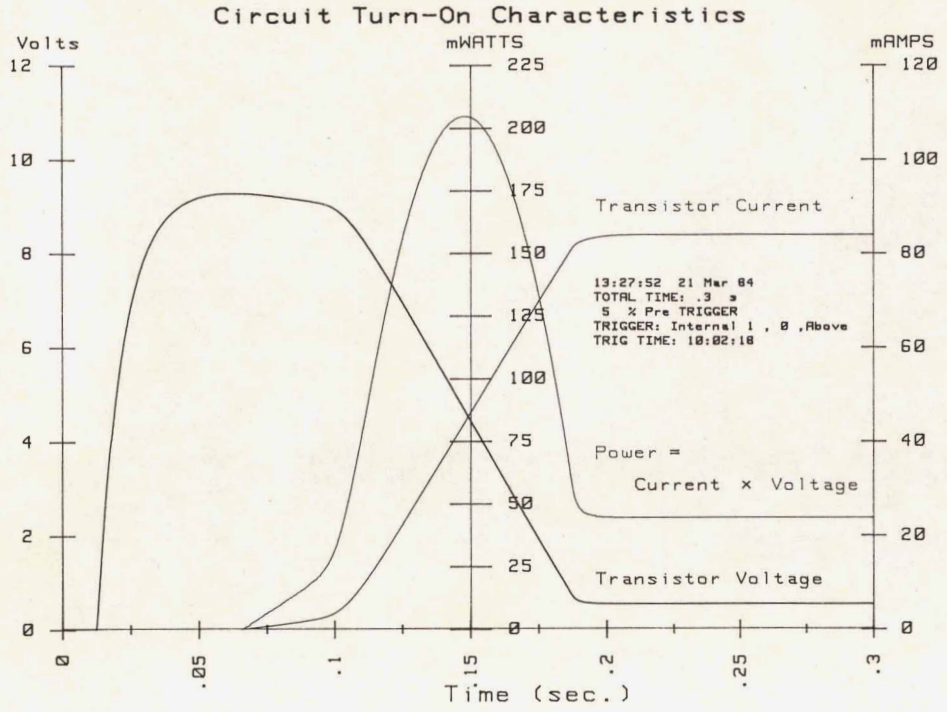


Fig. 6. *Example MGS plot showing a calculated parameter (power) versus measured current and voltage.*

Measurement Module

The measurement subsystem implements the ability to take measurements. It starts the measurement, and when data becomes available, receives the data and stores it in the software's channel buffers. There are three types of measurements: direct on-screen, direct on-paper, and data streaming. In direct on-screen measurements, the data is plotted to the screen in real time as the data is being stored in the software's channel buffers. Direct on-paper measurements emulate a traditional X-Y recorder and no data is sent to the computer. Data streaming mode allows up to ten thousand samples from each of three channels to be buffered into memory and then written out to a disc file for later processing.

Display Module

The display subsystem allows measurements and annotation to be displayed on the screen or on paper. There is a display setup mode that allows the user to specify which data channels of the measurement will be displayed. The display scale and the size of the displayed measurement can be adjusted. The output to paper can be formatted so that up to four measurements and their data can be plotted on one page.

Data Manipulation Module

In a measurement system the user may have a need to postprocess the recorded measurement. This module gives the user the capability of performing arithmetic operations on data channels. This subsystem has the capability of performing +, −, ×, ÷, square root, square, log, and negation. This subsystem gives the user the capability of building algebraic equations with two operands and one operator. Operands can be any data channel or constants or the result of the previous operation. The results can be displayed using the display module. The last five operations are shown in a small window. This is done to simplify the task of computing complex equations through the chaining of operations. For example, when measuring voltage and current, the subsystem can be used to compute power by multiplying the voltage and current readings as shown in Fig. 6. Manipulations not provided directly by the software can be applied to the data sets through user-written programs.

Storage and Retrieval Module

The storage and retrieval subsystem allows the user to save not only the measurement data but also the current measurement setup, annotation, and display setup parameters. When retrieving data, the users can select subsets of the data to be retrieved. For instance, the annotation can be stored from one measurement and retrieved into another. The measurement setup parameters will always be retrieved along with the measurement data because the data itself does not have meaning without its setup conditions. There is a file header at the beginning that contains information about where the data and setup parmeters are located in the file.

Annotation Module

The annotation subsystem gives the measurement graphics user the capability to put grids, axes, labels, lines, markers, and an annotation box on the measurement graph. It is not intended to do general-purpose drawing or to be a graphics editor. Some features are:

- Axes and grids feature automatic tic labeling in the units of the measurement. Log axes and grids are also available.
- Labels are useful for titles and for adding documentation to the graph. They can be added, changed, or deleted at will.
- Lines can be used for simple drawing.
- Markers annotate points on the data line and they can be automatically labeled with their respective x and y coordinates. The cursor can be used to step through points on the data line to position the marker.

- The annotation box can be used to supply information about the measurement, such as channel settings, trigger level, trigger time, and time of day.

Acknowledgments

In addition to the authors, Diane Fisher and Irene Ortiz contributed to the design and development of the MGS package. Larry Hennessee managed the project.

Analog Channel for a Low-Frequency Waveform Recorder

by Jorge Sanchez

THE ANALOG CHANNEL of the HP 7090A Measurement Plotting System conditions and digitizes the signals connected to the inputs of the instrument. The analog signals are amplified, filtered, and digitized by a series of stages as shown in Fig. 1. After the signals are digitized, the equivalent binary words are processed through a series of calibration procedures performed by the microprocessor to provide the full dc accuracy of the machine. The architecture of the channel is designed with flexibility of operation as a goal. Thus, the microprocessor is used to set up the multiple stages for coarse and fine gains and offsets. This allows the execution of zeroing and calibration routines and eliminates manual adjustments in the manufacturing process. (No potentiometers were used in the design. See box on page 22.) The analog channel has floating, guarded inputs. Through the use of isolation and shielding, common mode rejections of >140 dB for dc and >100 dB for 60 Hz are obtained.

Preamplifier

The analog channel preamplifier (Fig. 2) uses a set of low-noise, low-leakage JFETs and low-thermal-EMF relays to switch the inputs of amplifier A1 to the gain and attenuation string of resistors. The amplifier switches are connected in such a way as to set the 14 major ranges for the HP 7090A. (Other ranges are provided by a postamplifier as will be explained later.) The ranges are set by the microprocessor's loading the appropriate words in front-end registers 1 and 2. Amplifier A2 is used as a buffer to drive three different circuits:

- The internal guards that minimize printed circuit board leakage in critical areas
- The on/off and biasing circuits for the range setting switches (as set by front-end registers 1 and 2)

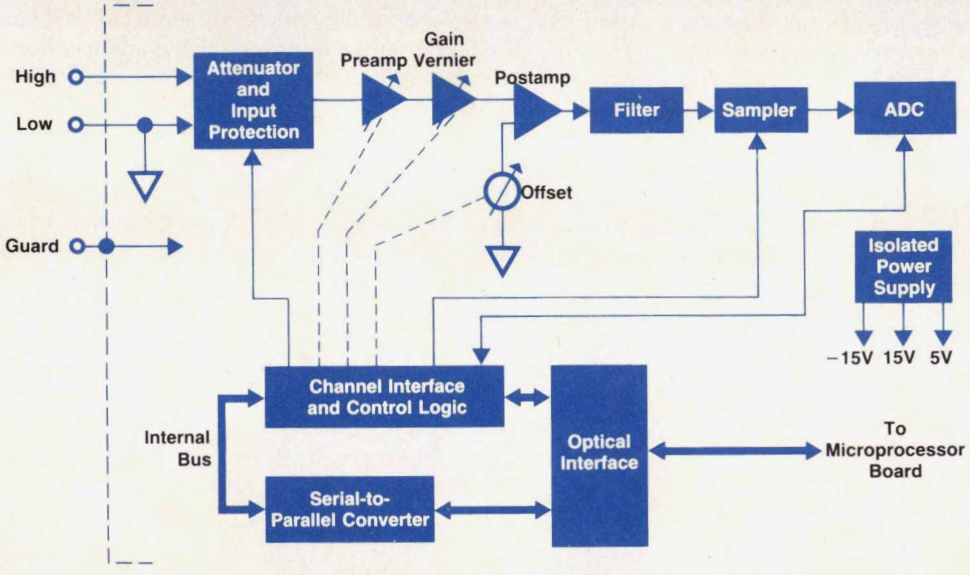


Fig. 1. *Block diagram of analog channel.*

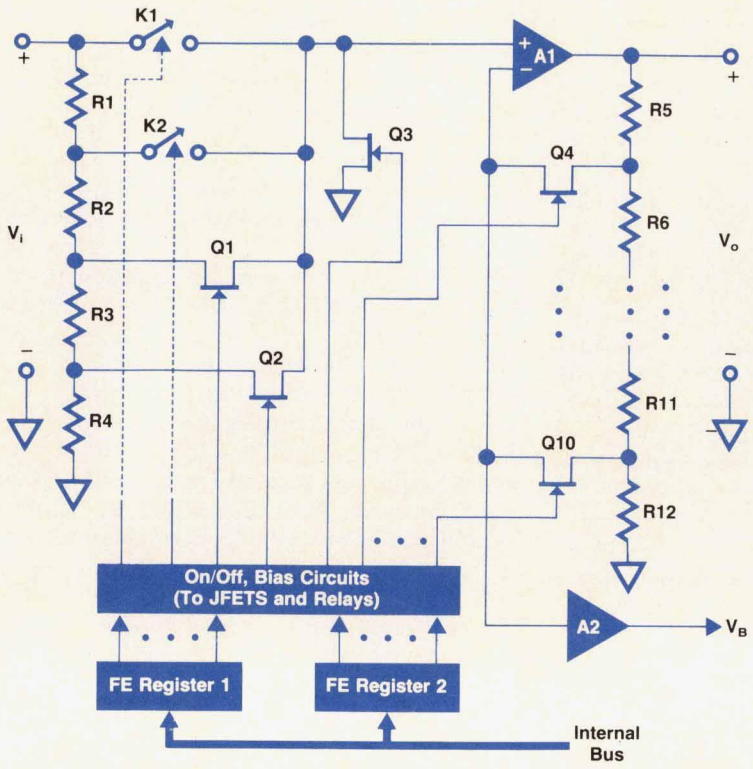


Fig. 2. *Analog channel preamplifier.*

- The input protection feedback loop.

To satisfy the performance requirements of the HP 7090A to handle signals as low as a few microvolts with a bandwidth spanning from dc to a few kilohertz, the design uses carefully chosen components such as the precision low-noise amplifiers A1 and A2 and metal-film resistors of small values (to avoid white noise). In addition, printed circuit board layout becomes critical. Hence, extensive use of guarding and shielding of critical areas is done, including the use of Teflon™ cups for the input node.

Transistor Q3 is part of the circuitry used in an autozeroing routine to eliminate channel offsets caused by initial component errors, temperature drift, and aging.

ESD and Overload Protection

Front-end inputs are likely to experience ESD (electrostatic discharge) transients since they can be touched by the user. Also, in a general-purpose instrument, temporary dc overloads may be applied. For this reason, protection circuits are necessary. Very often these circuits tend to degrade amplifier performance. This situation was avoided in the HP 7090A by using the circuit shown in Fig. 3.

If there is no way to prevent ESD from penetrating the machine, the next best thing is to shunt the transient to ground through a preferential path of impedance lower than the rest of the circuits. The primary ESD clamp is actuated by electron tube E1 and the source inductance. E1 has a very large resistance and low capacitance when in the off state. Hence, it does not degrade the amplifier's input impedance. Capacitor C1 turns off E1 after the surge. Resistor R1 discharges C1. This circuit can only limit V1 to several hundred volts because of the insufficient speed of E1.

The secondary protection devices clamp the input to a

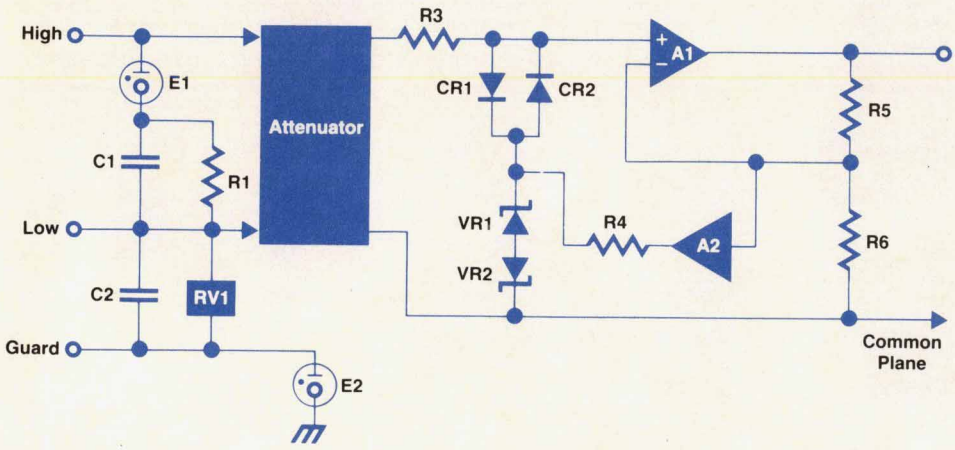


Fig. 3. *Input protection circuitry.*

voltage less than the maximum allowable voltage for A1. (This is also used as the dc overload protection.) The other circuits minimize leakages. Buffer A2 sets rectifiers CR1 and CR2 to zero bias by feedback. Leakage caused by Zener diodes VR1 and VR2 is provided by A2 and not by the minus node of A1.

To avoid sourcing current for the bottom plate of C1 from the common plane, and since there is no way to obtain a simultaneous turn-on of E1 and E2, C2 is installed between the low and guard terminals to provide the current.

RV1 is a voltage clamp device used to protect the devices between the low and guard terminals against overloads between the input terminals or against transients applied to the low terminal. The final shunting of the ESD transient to earth ground is provided by electron tube E2.

In a circuit such as this, care must be taken to shield or orient the components and connections to prevent reradiation of noise to other areas. In addition, the breakdown voltages of interconnections should be much higher than the breakdown voltages of the devices used. These protection circuits proved successful during testing by enduring many thousands of electrostatic discharges up to 25 kV that were applied to the inputs.

Vernier Gain Stage

The digitally programmable vernier stage consists of a 12-bit multiplying digital-to-analog converter (DAC) and an operational amplifier. Its main function, in conjunction with the preamplifier, is to provide the numerous calibrated ranges of the machine. The gain in this stage is represented by $G = -D/4096$, where D is the decimal equivalent of the binary word that is applied to the DAC. The number D is equal to the product of two scaling factors D1 and D2. D1 accounts for the vernier gain. It is derived from the range entered by the user and from internal routines in the microprocessor as indicated by the channel's range calibration equations. D2 is a fixed attenuation factor and is used as a coarse gain adjustment to account for system gain error caused by component tolerances.

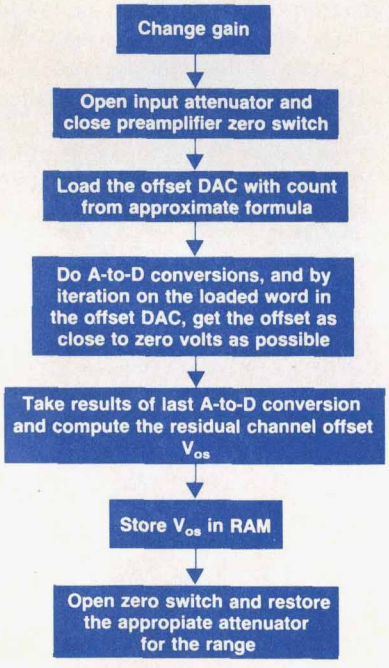


Fig. 4. *Flowchart of offset calibration routine.*

Postamplifier

The postamplifier stage has the following functions:

- It amplifies the signal to a voltage level that is suitable for the digitizer
- It contains a 3-kHz low-pass active filter
- It provides an offset voltage that is programmable by the microprocessor.

The programmable offset is accomplished by the use of a low-cost DAC. This converter is used primarily for subtracting out the analog channel's subsystem offset each time the ranges are changed, and for periodically performing a zero calibration to account for drifts. The offset DAC performs a coarse offset subtraction in hardware. To accomplish a fine offset calibration, the residual offset V_{os} is first found by the offset calibration routine (see Fig. 4). This offset is subtracted from the incoming data during the data correction routine, which is executed after the input signal is sampled.

A-to-D Conversion Circuits

This section consists of one sampling stage with two sample-and-hold devices connected in parallel and requiring an analog multiplexer, buffer and control logic, and a 12-bit analog-to-digital converter (ADC). Two sample-and-hold ICs are used here to be able to perform an A-to-D conversion on a sample while simultaneously acquiring the next sample (see Fig. 1 on page 22). After the conversion is completed, the sample-and-hold stages are swapped by the sequencing circuits and the cycle is restarted. This eliminates the acquisition time wait for a conversion cycle, thereby allowing the use of a slower low-cost converter.

Studies have shown that the eye can distinguish very small fluctuations in a ramp waveform when it is plotted. For this reason, a 12-bit-resolution ADC had to be used, since the HP 7090A can plot the digitized waveform.

Common Mode Rejection Ratio (CMRR)

The CMRR specifications of the HP 7090A demand a high degree of isolation between the analog channel and ground. This requires resistances on the order of gigohms and a maximum capacitance to ground of about 25 picofarads. There are two main areas that provide the isolation—the optical interface and the channel power supply.

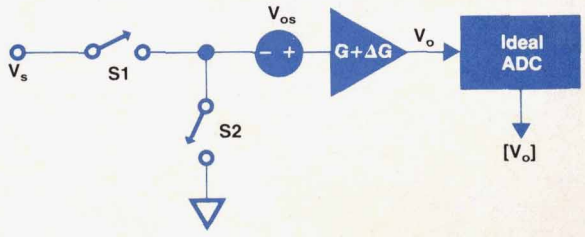


Fig. 5. *Simplified error model for HP 7090A front end.*

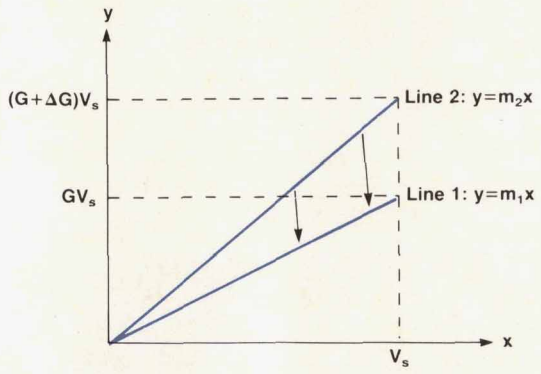


Fig. 6. *To calibrate gain, the response represented by line 2 is mapped into the ideal response indicated by line 1.*

The optical isolators provide all of the digital communications with the system processor in serial form. This is done with a small degradation in capacitance and resistance to ground. In addition, internal shields in the optocouplers provide common mode transient rejection of at least 1000 V/μs.

The most critical component in the channel power supply is the isolation transformer. To obtain a low isolation capacitance, three box shields are used. It can be demonstrated that three box shields will eliminate the most common ground loops associated with a floating front end.[1] Box shields and special manufacturing techniques minimize and cancel currents induced by the transformer into the preamplifier circuits. With this and careful pin assignments, low coupling capacitances in the hundreds of femtofarads are obtained.

The analog channel printed circuit board is placed in a sheet-metal shield to decrease coupling capacitances to ground and to minimize external source interference into the sensitive amplifiers.

Modern analog front ends often include digital and analog signals in the same set of circuits. This can become troublesome when there is a need to handle microvolt-level signals at high accuracy and wide bandwidths. Detailed attention to printed circuit board layout makes it possible to obtain high-quality signal conditioning. For this purpose, isolation of internal grounds and of analog and digital signals was done. Ground planes are also used to minimize intersignal capacitances. In addition, well-known techniques[2] are used throughout the board for isolating power supply output impedances and ground returns from the different stages.

Computer Calibration

To preserve accuracy under different temperature conditions and to compensate for the aging of components, the HP 7090A's microprocessor executes a series of calibration routines. These same routines allow the use of automated gain calibration at the factory. The calibration factors thus obtained are stored in a nonvolatile memory in the HP 7090A.

Every stage in the front end adds errors to the signal. The procedure followed is to lump all errors, refer them to the inputs, and separate them into gain and offset errors. Fig. 5 shows a simplified example of an error model. In this case G = ideal gain, V_s = input signal, ΔG = gain error, V_o = signal at the ADC, V_{os} = offset error, and $[V_o]$ = quantized value of V_o.

To calibrate the sampled signal, we first sample the system offset by closing S2 and opening S1. This is done in the HP 7090A during the offset calibration routine outlined in Fig. 4. This yields:

$$V_{o_1} = (G+\Delta G)\, V_{os}$$

Then, we acquire the input signal by opening S2 and closing S1, which gives:

$$V_{o_2} = GV_s + \Delta GV_s + GV_{os} + \Delta GV_{os}$$

After offset compensation we get:

$$V_{o_3} = V_{o_2} - V_{o_1} = GV_s + \Delta GV_s$$

To do a gain calibration, we map response line 2 in Fig. 6 into line 1 by the procedure explained in the box on page 22. This yields the gain calibration factor $G/(G+\Delta G)$. This factor is obtained for each one of the 14 major ranges of the machine. As mentioned before, these factors are stored in the HP 7090A's internal nonvolatile memory.

Accuracy in other ranges that use the vernier is guaranteed by the circuit design.

The gain calibration requires a final multiplication:

$$V_{o_4} = V_{o_3}(G/(G+\Delta G)) = [V_s(G+\Delta G)][G/(G+\Delta G)] = GV_s$$

This last quantity is indeed the amplified input voltage, which is the desired quantity.

Other more complex models, similar to the one above, are used to account for other operations of the machine such as user's entered offset, factory calibration routines, and combinations of interacting errors. The exact equations used for the corrections in firmware are also in a quantized form.

References

1. R. Morrison, *Grounding and Shielding Techniques in Instrumentation*, second edition, John Wiley & Sons, 1967.
2. Henry Ott, *Noise Reduction Techniques in Electronic Systems*, John Wiley & Sons, 1976.

Usability Testing: A Valuable Tool for PC Design

by Daniel B. Harrington

Evaluating the experiences of users unfamiliar with a new computer product can provide valuable guidance to the designer and the documentation preparer.

A KEY ELEMENT IN THE DESIGN of a personal computer is how easy it is for a new owner to set it up, get it running, and do basic tasks such as printing output, loading software, entering data, and handling files. To evaluate these qualities, HP's Portable Computer Division has conducted three usability tests, two on the Integral PC (one before, one after introduction) and one on The Portable (after introduction). A single test program uses ten reviewers, one per day, each performing for pay the same set of tasks on the selected computer model. The tasks are performed in the testing room at the division.

The reviewers are selected to meet the profile of the expected buyer of the computer. Each reviewer's experience is videotaped, and an observer in the test room constantly monitors the reviewer's progress (see Fig. 1). When a reviewer becomes frustrated enough to call the dealer for help, the observer acts as the dealer and offers the help requested. Product engineers and management are invited to observe the test sessions. The results of the test, including suggestions for product improvement, are widely distributed. Finally, a reviewer debriefing meeting is held where the reviewers and HP engineers can discuss the usability of the product.

Why Have Usability Testing?

Hewlett-Packard is committed to quality and customer satisfaction. To know if we're satisfying our customers, we must measure our performance. Usability testing provides one means of measuring product quality and customer satisfaction. This method has several advantages:

- Product engineers can observe users (the reviewers) using their products, both during product development and after market introduction. Tests conducted during product development allow changes in the design of the product to satisfy the observed needs of users.
- It's a controlled measurement allowing statistical evaluation and comparisons of user satisfaction before and after product changes are made.
- Product engineers can meet the group of reviewers at a debriefing meeting. At this meeting, engineers can hear what the reviewers liked and did not like about the product, and the product changes they wish HP would make. This meeting also allows dialog between engineers and reviewers.
- It's an especially effective test of documentation, a key part of this type of product.

Many of our competitors emphasize the human interface. They understand that buying decisions are affected both by the reported length of time it takes new users to get familiar with a computer and the difficulties users have encountered in using it. Corporate buying decisions are especially influenced by the computer productivity expected from a particular brand or model.

Magazine evaluations also focus on user-friendliness. Perhaps you've read, as we have, magazine reviews of new computers, in which the writers take great pleasure in describing their frustrations in trying to use the computers. Such negative reviews must hurt sales, just as positive reviews must help sales.

Customers do not like to be frustrated by incomprehensible error messages, manual jargon, confusing instructions, peripherals that won't work when connected, and all the other problems that a first-time user of a personal computer too often encounters. Usability testing offers an effective way to measure and reduce such problems.

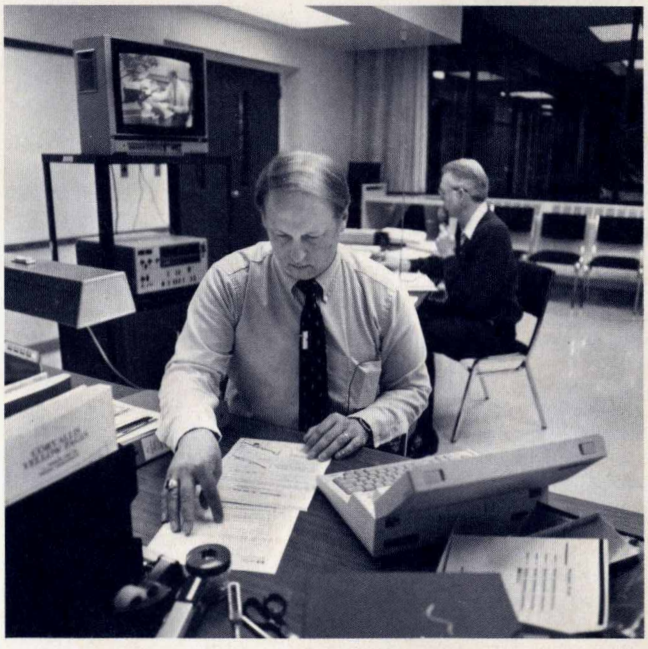


Fig. 1. *A reviewer studies the instructions for the computer being tested. Note the observer and monitor in background.*

How is Usability Testing Done?

We learn from product management the profile of the expected buyer of the computer we're about to test. We then seek people in the local community who fit that profile and who are not HP employees. We find most of them are excited about spending a day with us playing with a new computer. As a token of our appreciation, we pay reviewers for their help.

We encourage HP people to observe the usability test. We want those responsible for the product to watch and listen to these reviewers as they work. While it can be a humbling experience to see how the results of our efforts somehow fail to work in the reviewer's hands as we intended them to, such experiences are vital to developing a product that satisfies users.

Each reviewer spends a day using the computer in a simulated work environment. We equip the testing room with a table set up like a typical office desk, complete with plant and in-basket. At best, the test situation in which the reviewers find themselves is still foreign, but we try to create an atmosphere that is at least partially familiar. We feel the closer the testing environment is to a typical user's workplace, the more valid our results will be.

An opening questionnaire gives us the reviewer's computer experience and educational background. This information helps us qualify each reviewer's experiences during the test session. This questionnaire also confirms that the reviewer meets the profile of the expected buyer.

Before users operate a computer for the first time, most have studied the market and already know something about the particular computer they have chosen. Reading brochures and reviews, having discussions with dealers and other users, and watching others use the computer allow a user to set up and run a new computer more efficiently than one who has never seen nor heard of the product before opening the box. We can't completely duplicate this knowledge, especially for a product still under development, but we do give each reviewer a description of the product before the test session begins. For a released product, we mail a brochure and data sheet to each reviewer a week before the test starts.

The reviewers start with the computer in its shipping carton. We give each of them the same set of tasks or objectives, and ask them to perform them in any order they desire.

A video and audio recording of each session is made. These recordings serve several purposes:

- They support the notes the observer makes at each session.
- They are available for study after the test is over.
- They provide the raw material for the summary tape shown at the reviewer debriefing meeting.

We urge reviewers to comment freely. The audio portion of the tape is often the most important. We want reviewers to tell us what they're doing, how they feel, what they like and don't like about the product; in short, we want almost a stream-of-consciousness narrative.

An observer is always in the room with the reviewer. The observer uses notes taken during the usability test to write the test report. When the observer needs more opinions and information from the reviewer, the reviewer is asked appropriate questions during the test.

When we started these tests, we were concerned about the observer sharing the test room with the reviewer. The standard testing arrangement used by IBM[1] consists of two rooms separated by a one-way mirror. The reviewer is alone in one room, which is identical to a typical office. The observers, video cameras, and other equipment are in the other room. We started with and still use only one room, but we feared the observer's presence would inhibit the reviewer's actions and comments, making the results less valid. Therefore, we specifically asked reviewers who helped us with our first test if the observer's presence hurt the effectiveness of the test. They told us the nearness of the observer helped, rather than hurt the process. They felt they were talking to a human rather than a machine, which made it easier to comment freely. They also appreciated the reviewer's encouragement and requests for comments.

We also emphasize that the product is on trial, that the reviewer cannot fail. It's important that reviewers feel at ease so that their experiences are as close as possible to those real users would experience. However, some reviewers still feel under some pressure to perform, and try to finish the tasks as fast as they can to do a good job. An observer can help reduce this pressure by creating an atmosphere of you-can't-fail informality. This is another advantage in having the observer share the test room with the reviewer.

The reviewers have only two sources of help:

- The manuals, disc-based tutors, on-screen help messages, and other material delivered with the product.
- Their dealer (the observer).

Reviewers that reach a level of frustration that would produce a call to their dealer if they were using their own computer in their home or office can pick up the unconnected phone on their desk. This action tells the observer that a dealer call is being made. The observer then acts as the dealer and gives whatever help is needed. The number

Fig. 2. *HP's Integral Personal Computer[2] is a powerful multitasking computer system in a 25-lb transportable package. Designed for technical professionals, it features a built-in printer, display, disc drive, and HP-IB interface and the HP-UX operating system, HP's version of AT&T Bell Laboratories' UNIX™ operating system.*

of such calls and the reasons for them can tell us a lot about what product features are hard to understand or not working well.

A closing questionnaire asks for opinions about the product. In general, this questionnaire asks two types of questions. One type asks reviewers to rank their level of agreement or disagreement with a number of positive statements about various features of the product, such as:

The owner's manual is easy to understand.

The error messages are easy to understand.

I like the display.

Each reviewer is asked to rank each statement from 1 (strongly agree) to 5 (strongly disagree). The other general type of question asks reviewers to comment on various parts of the product, such as manuals, keyboard, display, help messages, etc. Often, a product feature like a manual is the subject of both a ranking question and an essay question. Another common question asks reviewers to identify the most difficult or the three most difficult tasks. That question is followed with a ranking question something like this: "Considering the difficulty of the task you identified as the most difficult, the instructions for that task are as clear as they can be."

The video recorder is stopped while the closing questionnaire is completed. Then it is turned on again to record the closing interview. The observer chooses some closing topics to discuss further, generally about product areas reviewers felt needed improvement. These interviews often produce some of the best and most useful video footage.

About two weeks after the last test session, the reviewers and the product engineers meet together. This is a very useful meeting. It allows the product engineers (hardware, software, electronic, system, packaging, manual, quality, production, etc.), management, and anyone else who is interested to hear reviewers' opinions directly. By asking questions, the audience can draw out additional reviewer opinions and suggestions.

The final report is widely distributed. This report describes the test and gives the reviewers' opinions and suggestions.

How Has Usability Testing Helped?

During the preintroduction test of the Integral PC,[2] reviewers felt the initial mechanical design did not give an impression of quality and ruggedness. A description of this computer will help to explain their complaint. The Integral PC (Fig. 2) is a transportable computer. The bottom of the keyboard is the front face of the closed-up computer, and the carrying handle is attached to the top, which opens up and folds back to release the keyboard and reveal the built-in flat-panel display, 3½-inch disc drive, and ThinkJet printer. The main reviewer complaint about the apparent lack of ruggedness centered on the mechanism that controls the opening and closing action of the top cover. This mechanism had been tested by engineering and had satisfied their tough strength specifications. However, the reviewers felt the looseness of the mechanism suggested weakness and sloppy design.

The mechanical engineers accepted the reviewers' judgment that the top cover mechanism should not only be rugged, but should also appear rugged. They made design changes that largely eliminated the looseness of this mechanism, and the postintroduction usability test of the Integral PC told us that they did an excellent job. The reviewers who judged this computer during this second test felt the computer did give an impression of quality and ruggedness.

The Integral PC's on-screen tutor, a new type of instruction product for our division, incorporated usability testing as a key item in its development schedule. The strong positive acceptance of the final tutor would not have been possible without the user feedback given by two informal usability tests and a final, formal usability test conducted during product development.

The Integral PC Setup Guide (Fig. 3) is another new type of instruction product for our division. This guide uses a series of pictures with very few words to tell a first-time user how to open the computer's case, connect the keyboard and optional mouse, and start the on-screen tutor. Other sections of this setup guide tell the user how to install the printhead cartridge for the built-in ThinkJet printer, how to load fanfold paper into the printer, and how to prepare the Integral PC for transporting.

Usability testing was incorporated into the development schedule for this setup guide. These tests indicated the need for major changes in the initial guide. The postintroduction usability test proved the final setup guide was very useful, and suggested some further improvements.

The preintroduction usability test of the Integral PC suggested improvements in the packaging. The initial shipping carton design we tested included a thin, flat parts box inside the shipping carton. Either of the two large faces of this parts box could be opened easily by users, but the box would reveal all of its contents only when one of these faces was opened. If the other face was opened, many of the smaller parts were well hidden. When the reviewers pulled this parts box out of the shipping carton, chance would dictate which large face was up when the box was laid on a table. If the wrong side faced up, the wrong side was opened, and parts were lost.

The packaging engineer observed some of the reviewers opening the wrong side, and had a cure specified before

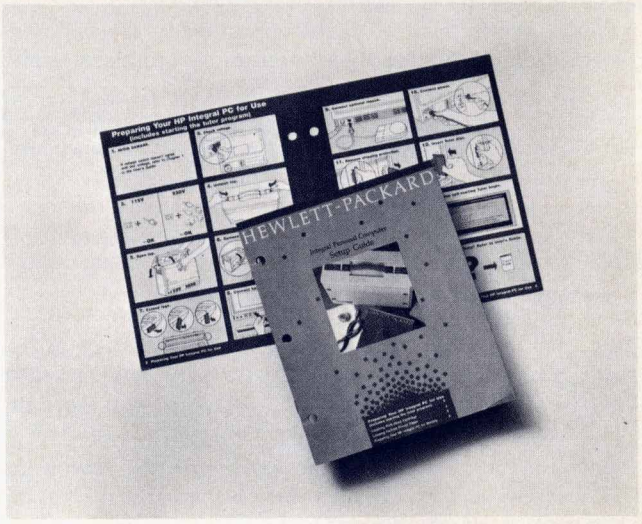


Fig. 3. *Integral PC Setup Guide, a 10-page guide whose development depended on usability testing.*

the sequence of usability tests was over. He specified that the words "Open this side" appear in large letters on the right side, and the words, "Open other side" appear in large letters on the wrong side. This improved parts box was tested during the postintroduction usability test. During this test, the reviewers proved that people often don't see what they look at and don't read what they see. In spite of the words "Open other side" printed in large letters on the wrong side of the parts box, several reviewers opened the wrong side anyway, and did not see or remove the smaller parts, including the ink cartridge for the ThinkJet Printer. One reviewer suggested that we design our parts box to open only one way. Again the packaging engineer responded quickly, and the Integral PC parts box now opens only one way. This example shows the importance of testing the cures inspired by previous tests.

During the preintroduction test of the Integral PC, reviewers felt the disc drive busy light was too dim. Engineering responded, and the production computer now has a satisfyingly bright light to indicate disc drive activity.

Some help screens provided by The Portable (Fig. 4), and displayed by pressing a function key, do not state clearly how to get out of the help screen. One help screen set, consisting of a number of screens, does not tell the user how to exit until the fourth screen. The software group in engineering listened to the reviewer's comments about this. The Portable PLUS, developed after The Portable, also uses help screens, but the first screen of every help screen set clearly tells the user how to exit.

The Portable includes a disc-based diagnostic program. This program was loaded into the memory of the first shipped units of The Portable, and its label was shown in the PAM's (Personal Application Manager's) main screen at the far left. When The Portable's display was first turned on, the selection arrow pointed to the diagnostic program's label. During the usability test, several reviewers pressed Start on the keyboard to see what would happen. This would start the diagnostic program, causing much confusion. Again engineering listened, and they specified that this disc-based diagnostic program no longer be loaded into The Portable before shipment, although the disc containing this program continues to be included with the product.

The Portable was the first computer from this division to use three-ring binders for its manuals. We elected to put five separate manuals into one binder separated by tabs, since these five manuals fit comfortably in one binder, and doing so reduced product cost. A second binder was used to contain only one manual, the *Lotus™ 1-2-3™ User's Manual*. Even though we stated clearly (we thought) on the second page of the first manual that five separate manuals were in the binder, and gave descriptions of each, many reviewers were confused. They thought instead that the binder contained several sections of one manual. For example, they would look in the index of the last manual, the MS™-DOS *Operating System User's Guide*, for page references to the other manuals. Since each of the five manuals started with page 1-1, reviewers were understandably frustrated. As a result, future loose-leaf binders will each contain only one loose-leaf manual, or will provide clear ways for users to realize that it contains more than one.

The Portable reviewers made many other suggestions for manual improvement. Three of the more important suggestions that have been implemented are:

- Start each chapter with a table of contents.
- Every reference to a function key should be followed with the keycap label, like Start (**f1**).
- Every keystroke sequence that requires pressing **Return** to generate the desired action should include **Return** as the last keystroke.

The postintroduction test of the Integral PC gave us our first chance to test the general manual improvements. Each reviewer opened a new box fresh from the production line to ensure that the contents were arranged and packaged just as actual users would see them when opening their newly purchased computer. One complaint these reviewers had was the difficulty and frustration of tearing the plastic shrink wrapping off the manual binders. They were especially vocal about the very rugged clear plastic we used for the plastic bag containing the setup guide and tutor disc. These reviewers suggested we add an easy-open tab to the shrink wrapping and use a zip-lock plastic bag for the setup guide and tutor disc. These suggestions are being considered.

Our documentation department maintains a revision file on all current manuals. When a manual reprinting becomes due, the responsible writer checks the appropriate file and incorporates the corrections and changes that have collected since the last printing. All reviewer suggestions for manual improvements made during the postintroduction test of the Integral PC have been inserted in the appropriate manual revision file, provided the suggestions make sense (most of them do). In this way, the next printing of each manual will profit from the feedback given to us by these reviewers.

What Improvements Have We Made to the Testing Process?

Each time we conduct a usability test we learn how we can improve it further. Some of the improvements we've made to the testing process since we began are:

- The task list we used for the early tests was quite detailed.

Fig. 4. *The Portable is a 9-lb personal computer with built-in software for file management, spreadsheets, graphics, word processing, and data communications.*

For instance, the test of The Portable asked each reviewer to perform 38 narrowly defined tasks that we expected reviewers to perform in a particular order. For example, the first task asked them to turn on the computer. We now ask reviewers to complete a smaller series of broader objectives, and urge them to complete these objectives in any logical order. (An example of an illogical order would be to start a program from the electronic disc before first copying that program from a flexible disc.) The first task listed on our latest 14-item task list asks reviewers to install extra memory, but since we urged reviewers to perform tasks in any order, one reviewer performed this task near the end of his session.

- In the beginning, we used only one microphone, a lapel mike for the reviewer. Therefore, only half of the several conversations per session between the observer and the reviewer were recorded. Now the observer also has a mike, and we use a mixer to feed both audio signals to the video recorder.
- The videotape of the first test consisted exclusively of medium-to-long-distance shots of the reviewer working at the desk. Much of the recorded action consisted of reviewers turning manual pages hoping to find answers to their problems. Now we only use long-distance shots to show the action during unpacking, connecting peripherals, loading printer paper, etc. As soon as a reviewer starts working at the keyboard, we record a close-up shot of the display. The main advantage is that the observer can tell what the reviewer is doing by watching the computer's display in the TV monitor.
- We now record the closing interview, rather than simply take notes as we did at first. These produce some of our best recordings, since they often contain excellent useful comments on our products.
- We have always held debriefing meetings, in which the reviewers have a chance to give their opinions directly to the people in the division responsible for the product. We now have added another feature to these meetings—a special videotape lasting one hour or less and containing the most significant results of the approximately 50 hours of videotape recorded during the 10 sessions. These have proved quite informative, and clearly show the occasional sad and funny experiences of new users when they're confronted with the result of our work.
- During preintroduction tests, serious and obvious product and manual errors are now corrected immediately during the test program where possible. This allows us to measure these cures during the later sessions of the same test, permitting further change if needed before product release.

Acknowledgments

The author thanks Bob Ulery, manager of the Portable Computer Division's product support department, Don Cole, the document and training department manager, and John Beaton, the author's supervisor during the time this work was done. Their initiative and continuing strong support makes our usability testing program possible.

References

1. J. Gallant, "Behind the Looking Glass, Users Test IBM Software," *Computerworld*, February 27, 1984, p.4.
2. Complete issue, *Hewlett-Packard Journal*, Vol. 36, no. 10, October 1985.

Hewlett-Packard Company, 3000 Hanover Street, Palo Alto, California 94304

HEWLETT-PACKARD JOURNAL

January 1986 Volume 37 • Number 1

Technical Information from the Laboratories of Hewlett-Packard Company

Hewlett-Packard Company, 3000 Hanover Street
Palo Alto, California 94304 U.S.A.
Hewlett-Packard Central Mailing Department
P.O. Box 529, Startbaan 16
1180 AM Amstelveen, The Netherlands
Yokogawa-Hewlett-Packard Ltd., Suginami-Ku Tokyo 168 Japan
Hewlett-Packard (Canada) Ltd.
6877 Goreway Drive, Mississauga, Ontario L4V 1M8 Canada

Bulk Rate
U.S. Postage
Paid
Hewlett-Packard
Company

5953-8543