Proceedings

1982 INTERNATIONAL MEETING

HP3000 IUG

SAN ANTONIO ★ FEB 28 - MARCH 5

TABLE OF CONTENTS

AUTHOR INDEX

	Class	No.
Carnes, Lance	5	79
Chi, Cecile	7	55
Cohen, Jon	3	81
Collins, Melissa J.	11	26
Colwell, Nancy	2	52
Crow, William M.	7	49
Davis, Lloyd D.	11	28
Dennis, Steven J.	7	18
Dummer, David C.	2	53
Edmunds, David	3	86
Ellzey, Gavin L.	3	61
Fraser, Tom	4	36
Fratus, Pete	11	25
Galpin, Amy J.	1	42
Garvey, Robert B.	1	43
Geffken, Joachim	5	73
Goertz, Jason M.	1	58
	7	59
Green, Robert M.	2	5
	1	6
Greer, David J.	3	3
	4	4
Grossler, Jorg	3	15
	1	16
Harbron, Thomas R.	2	71
Heidner, Dennis	2	34
Holt, Wayne E.	1	42
Hulme, John	6	69
	1	70
Kolitz, Nancy	4	30
Kramer, Jim	5	57
Langenwalter, Gary L.	11	33
Leeper, Kim D.	3	2
	4	90
Lett, Robert	7	21
Lind, Janet	11	63
McAfee, Bill	3	31
Newcomer, Eric A.	5	65
Nichols and Company	7	72
Ow, Gary	4	1
Perrin, Steve	7	21
Raschka, James G, CPIM	7	24
Ray, John R.	11	28
Rebensburg, Klaus	4	13
Ridgway, Paul W.	3	62
Roland, Vince	1	48
Ryder, Giles	3	85
Saunders, Steven K.	4	27
Scavullo, Bob	7	80
Schulz, Duane	2	64
Scroggs, Ross	6	40
Somers, Peter	4	12
Stambaugh, Jan	2	74
	11	75
Straayer, Russell A.	11	38
Tibbetts, John	6	40
	11	41
Trasko, Mark S.	7	20
Utz, Walter J., Jr.	11	88
Volokh, Eugene	11	17
Weld, Gloria	11	84
Wheeler, Bruce	1	35
Winters, Craig	3	31
Wright, Steve	4	83

Overview of Optimizing (On-Line and Batch)

Robert M. Green
Robelle Consulting Ltd.

SUMMARY

The performance of many HP3000 installations can often be improved significantly. There are general principles for delivering better response time to on-line users, and other principles to speed execution of production batch jobs. As long as users continue to consumer the extra horsepower of new HP3000 models by loading them with new applications, there will continue to be a need for optimizing knowledge and tools. And, if interest rates remain at current levels, many managers may not be able to upgrade to faster computers as soon as they would like.

CONTENTS

SECTION 1
HOW TO IMPROVE ON-LINE RESPONSE TIME

I have identified five general principles which help in optimizing the performance of on-line programs:

- Make each disc access count.
- Maximize the value of each "transaction."
- Minimize the run-time program "size."
- Avoid constant demands for execution.
- Optimize for the common events.

On a systems programming project, such as a data entry package or a text editor, you should be able to apply all five of these principles with good results. That is because systems software usually deals with MPE directly and most of the sources of slow response are under your control. Applications software, on the other hand, usually depends heavily upon data management sub-systems such as IMAGE and V/3000. The optimizing principles proposed here may not be as easy to apply when so many of the causes of slow response are beyond your control. However, there are still many ways in which you can apply the guidelines to application systems (monitoring program size, designing your database and laying out your CRT screens). Relying upon standard software not only increases your programmer productivity, it also provides an unexpected bonus: any improvements that the vendor makes in the data management tools will immediately improve the efficiency of your entire application system, with no re-programming or explicit "optimizing" on your part.

I. A. Make Each Disc Access Acount

Disc accesses are the most critical resource on the HP3000. The system is capable of performing about 30 disc transfers per second, and they must be shared among many competing "consumers." (This can increase to 58 per second under the best circumstances, and can degrade to 24 per second when randomly accessing a large file.) MPE IV can double the maximum disc throughput for multi-spindle systems by doing "look-ahead" seeks, but only for the Series II/Series III, not the Series 30/33/44.

·The available disc accesses will be "spent" on several tasks:

- Virtual memory management (i.e., swapping).
- MPE housekeeping (logon, logoff, program load, etc.).
- Lineprinter spooling.
- Accesses to disc files and databases by user programs (the final payoff).

If the disc accesses are used up by overhead operations, there will not be sufficient left to provide quick response to on-line user transactions. Some examples of operations that consume disc accesses on the HP3000 are:

- Increasing the number of keys in a detail dataset, thus causing IMAGE to access an extra master dataset on each DBPUT. Also, making a field a key value means that a DBDELETE/DBPUT is required to change it (which is 10 times slower than a DBUPDATE).
- Increasing the program data stack by 5000 words, thus causing the MPE memory manager to perform extra, swapping disc accesses to find room in memory for the larger stack.
- Improperly segmenting the code of an active program, causing many absence traps to the memory manager to bring the code segments into main memory.
- Constantly logging on and off to switch accounts.
- Defining a database with a BLOCKMAX value of 2000 words, thus limiting IMAGE to about 13 data buffers in the extra data segment that is shared by all users of that database. With such a small number of buffers, there can be frequent buffer "thrashing." This effectively eliminates the benefits of record buffering for all users of the database, and greatly increases disc accessing.

Much of the remainder of this document is devoted to methods of "saving the precious resource — disc accesses."

I. B. Maximize the Value of Each "Transaction"

This principle used to read, "Maximize the Value of Each Terminal Read," but I have generalized it to "transaction" to take into account the prevalence of V/3000, DS, MTS and other "communications" tools. In the terms of MPE IV, a "transaction" begins when the user hits the 'return' key (or Enter) and ends when the user can type input characters again. This includes the time needed to read the fields from the terminal (or from another HP3000), to validate them, perform database lookups and updates, format and print the results, and issue the next "read" request.

Each time a program reads from the terminal, MPE suspends it and may swap it out of memory. When the operator hits the 'return' key, the input operation is terminated, and MPE must dispatch the user process again. If MPE has overlaid parts of the process, they must be swapped back into main memory again. Due to the overhead needed to dispatch a process, a process should get as much work done as possible before it suspends for the next terminal input.

The simplest way to program data entry applications is to prompt for and accept only one field of data at a time. This is also the least efficient way to do it. Since there is an unpredictable "pause" every time the user hits 'return' (depending upon the system load at the moment), consistently fast response cannot be guaranteed. The resulting delays are irritating to operators. They can never work up any input speed, because they never know when the computer is ready for the next input line. If response time and throughput are the only considerations, it is always preferable to keep the operator typing as long as possible before hitting the 'return' key. Multiple transactions per line should be allowed, with suitable separators, and multiple lines without a 'return' should be allowed. If you are using V/3000, the same principles applies: each high-volume transaction should be self-contained on a single form, rather than spread out over several different forms.

I. C. Minimize the Run-Time Program "Size"

The HP3000 is an ideal machine for optimizing because of the many hardware features available at run-time to minimize the effective size of the program. Even large application systems can be organized to consume only a small amount of main memory at any one time. Each executing process on the HP3000 consists of a single data segment called the "stack," several extra data segments for system storage, such as file buffers, and up to 63 code segments. All segments (code and data) are variable-length and can be swapped between disc and main memory.

Program code which is not logically segmented makes it harder for the memory manager to do its job, causing disc accesses to be used for unnecessary swaps. Proper code segmentation is a complex topic (more like an art than a science), but here is a simplified training course: write modular code; don't segment until you have 4000 words of code; isolate modules that seldom run; isolate modules that often run; aim for 4000 words per segment, and group modules by "time" rather than "function;" if you reach 63 segments, increase segment size, but keep active segments smaller than inactive ones.

Although every process is always executing in some code segment, the code segment does not belong to the process, because a single copy of the code is used by all processes that need it. Since code is shared, it does not increase as the number of users running a given program increases. Most of your optmizing should be directed to the data areas (which are duplicated for each user). A 3000 can provide good response to more terminals if most data segments are kept to a modest size (5000 to 10,000 words). To keep stacks small, declare most data variables "local" to each module (DYNAMIC in COBOL), and only use "global" storage (the mainline) for buffers and control values needed by all modules. Dynamic local storage is allocated on the top of the stack when the subroutine is entered, and is released automatically when the subroutine is left. This means that if the main program calls three large subroutines in succession, they all reuse the same space in the stack. The stack need only be large enough for the deepest nesting situation. By inserting explicit calls to the ZSIZE intrinsic, you can further reduce the average stack size of your program.

You can also minimize stack size by ensuring that constant data items (such as error messages and screen displays) are stored in code segments rather than in the data stack. Since constants are never modified, there is no logical reason that they should reside permanently in the data stack. By moving them to the code segment, one copy of them can be shared by all users running the program. In SPL, this is done by including =PB in a local array declaration or MOVEing a literal string into a buffer. In COBOL, constants can be moved to the code segment by DISPLAYing literal strings in place of declared data items. In FORTRAN, both FORMAT statements and DISPLAYed literals are stored in the code.

A frequently overlooked component of program "size" is the effect of calls to system subroutines (IMAGE, V/3000, etc.). These routines execute on the caller's stack, and the work they do is "charged" to the caller. In many simple on-line applications (dataset maintenance program, for example), 90% of the program's time and over 50% of the stack space will be controlled by IMAGE and V/3000. You should be aware of the likely impact of the calls that you make. Do you know how many disc accesses a particular call to DBPUT is going to consume? As an example of how ignoring the "extended size" of a program can impact response time, consider the following case:

An application with many functions can be implemented with one of two different strategies. The first, and simplest, strategy is to code the functions as separate programs and RUN them via a UDC (or CREATE them as son processes from a MENU program). Each function opens the databases (and formsfile, etc.) when you RUN it, and closes them before stopping.

The second strategy is to code each function as a subprogram that is passed in the previously opened databases (and forms-file, etc.) as a parameter from a mainline driver program. If the application requires frequent movement from function to function (performing only a few transactions in each function), the "process" strategy will be up to 100 times slower than the "subprogram" strategy. The resources required to RUN the programs, open the databases, close the databases, and perform other "overhead" operations will completely swamp the resources needed to perform the actual transactions.

I. D. Avoid Constant Demands for Execution

The HP3000 is a multi-programming, virtual-memory machine that depends for its effectiveness on a suitable mix of processes to execute. The physical size of code and data segments is only one factor in this "mix." The "size" of a program is not just the sum of its segment sizes; it is the product that results from multiplying physical size by the frequency and duration of demands for memory residence (i.e., how often, and for how long, the program executes). A given 3000 can support many more terminals if each one executes for one second every 30 seconds, rather than 60 seconds every two minutes. Each additional terminal that demands continuous execution (in high priority) makes it harder for MPE to respond quickly to the other terminals.

Here are some examples of the kind of operation that can destroy response time, if performed in high priority:

- EDIT/3000, a GATHER ALL of a 3000-line source file.
- V/3000, forms-file compiles done on four terminals at once.
- QUERY, a serial read of 100,000 records (or any application program that must read an entire dataset, because the required access path is not provided in the database).
- SORT, a sort of 50,000 records.
- COBOL, compiles done on four terminals at once.

You should first try to find a way to avoid these operations entirely. (Can you use QEDIT instead of EDIT/3000? Would a new search item in a dataset eliminate many serial searches, or could you use SUPRTOOL to reduce the search time? Are you compiling programs just to get a clean listing?)

After you have eliminated all of the "bad" operations that you can, the remainder should be banished to batch jobs that execute in lower priority (this works better in MPE IV than III). Since jobs can be "streamed" dynamically by programs, the on-line user can still request the high-overhead operations, but the system fulfills the request when it has the time. The major advantage of batch jobs is that they allow you to control the number of "bad" tasks that can run concurrently (set the JOB LIMIT to 1 for best terminal response).

I. E. Optimize for the Common Events

In any application where there is a large variation between the minimum and maximum load that a transaction can create, the program should be optimized around the most common size of transaction. If a program consists of 20 on-line functions, it is likely that four of them will be most frequently used. If so, your efforts should be directed toward optimizing these four functions; the other functions can be left as is. Because the HP3000 has code segmentation and dynamic stack allocation, it is possible for an efficient program to contain many inefficient modules, as long as these modules are seldom invoked.

Since MPE will be executing a great deal of the time, you should become competent at general system tuning. Learn to use TUNER, IOSTAT, and SYSINFO (and the new :TUNE command in MPE IV). Any improvement in the efficiency of the MPE "kernel" will improve the response time of all users.

You do not have infinite people-resources for optimizing, so you must focus your attention on the factors that will actually make a difference. There is no point in optimizing a program that is seldom run. The MPE logging facility collects a number of useful statistics that can be used to identify the commonly accessed programs and files on your system. Learn to use the contributed programs FILERPT and LOGDB (Orlando Swap). If you are using IMAGE transaction logging, the DBAUDIT/Robelle program will give you transaction totals by database, dataset, program, and user (total puts, deletes, updates, and opens). Such statistics help in isolating areas of concern.

You can optimize application programs around the average chain length for detail dataset paths (the contributed program DBLOADNG will give you this information). Suppose you need to process chains of entries from an IMAGE dataset. If your program only provides data buffers for a single entry, you will have to re-read each entry on the chain each time you need it (extra disc I/O!!). Or, if you provide room for the maximum chain length, the data stack will be larger than needed most of the time (the maximum chain length is often much larger than the average). The larger data stack may cause the system to overload, eliminating the benefits of keeping the records in your stack. You should provide space in the stack for slightly more than the average number of entries expected. This will optimize for the common event.

SECTION II
ON-LINE OPTIMIZING EXAMPLE: QEDIT

QEDIT is a text editor for the HP3000 that was developed by Robelle Consulting Ltd. The primary objective of QEDIT is to provide the fastest editing with the minimum system load. Other objectives include conservation of disc space, similarity to EDIT/3000 in command syntax, ability to recover the workfile following a system crash or program abort, and increased programmer productivity.

QEDIT is an alternative to a hardware upgrade for users who are doing program development on the same HP3000 that they are trying to use for on-line production. Every optimizing paper in recent years by an HP performance specialist has recommended avoiding EDIT/3000. They usually recommend the "textfile-masterfile" approach to program development. (You do not actually edit your source program; instead, you create a small "textfile" containing only the changes to your "masterfile," then merge the two files together at compile-time). QEDIT allows you to have "real" editing on your HP3000, with less overhead than the "textfile masterfile" method, and still give good response time to your end-user terminals.

II. A. QEDIT and "Disc Accesses"

In order to reduce disc accesses, QEDIT eliminates the overheads of the TEXT, KEEP and GATHER ALL commands of EDIT/3000. These three operations have the most drastic impact upon the response time of the other users. QEDIT attacks the problem of KEEPs by providing an interface library that fools the HP compilers into thinking that a QEDIT workfile is really a "card image" file. As a result, it is never necessary to KEEP a workfile before compiling it. Since KEEPs are rarely used, most TEXTs are eliminated. The LIST command was given the ability to display any file (e.g., /LIST DBRPT1.SOURCE), so that a TEXT would not be required just to look at a file. TEXT is only needed when you want to make a backup or duplicate copy of an existing file. Since most users choose to maintain their source code in QEDIT workfiles (they use less disc space), the TEXTing of workfiles is optimized (by using NOBUF, multi-record access) to be four to seven times faster than a normal TEXT of a card-image file. The GATHER ALL operation is slow because it makes a copy of the entire workfile in another file. QEDIT renumbers up to 12 times faster by doing without the file copy.

Disc accesses during interactive editing (add, delete, change, etc.) are minimized by packing as many contiguous lines as possible into each disc block. Leading and trailing blanks are removed from lines to save space. The resulting workfile is seldom over 50% of the size of a normal KEEP file, or 25% of the size of an EDIT/3000 K-file (workfile). Most QEDIT users maintain their source programs in workfile form, since this saves disc space, simplifies operations (there need be only one copy of each version of a source program), and provides optimum on-line performance.

QEDIT always accesses its workfile in NOBUF mode, and buffers all new lines in the data stack until a block is full before writing to the disc. Wherever possible in the coding of QEDIT, unnecessary disc transfers have been eliminated. For example, the workfile maintains only forward direction linkage pointers, which reduce the amount of disc I/O substantially. Results of a logging test show that reducing the size of the workfile and eliminating the need for TEXT/KEEP reduce disc accesses and CPU time by 70-90%.

II. B. QEDIT and "Transaction Value"

Like EDIT/3000, QEDIT allows either a single command per line (/ADD), or several commands on a line, separated by semi-colons (/LIST 5/10;M 6;D 5). The principle of maximizing transaction value has been applied with good results to the MODIFY command. In

EDIT/3000, several interactions may be needed to modify a line to your satisfaction. QEDIT allows you to perform as many character edits as you like on each transaction; many users can perform all of their changes in a single pass. For complex character editing, such as diagrams, version 3.0 of QEDIT will provide "visual" editing in block-mode.

II. C. QEDIT and "Program Size"

QEDIT is a comletely new program, written in highly structured and modular SPL. The code is carefully segmented, based on the knowledge of which SPL procedures are used together and most frequently. Only two code segments need be resident for basic editing, and the most common function (adding new lines) can be accomplished with only a single code segment present.

QEDIT uses a modest data stack (3200 words) and no extra data segments. The stack expands for certain commands (especially the MPE :HELP command), but QEDIT contracts it back to a normal size after these infrequent commands are done. All error messages are contained in the code, isolated in a separate code segment that need not be resident if you make no errors.

Use of CPU time is th eother dimension to program "size." QEDIT is written in efficient SPL and consumes only a small amount of CPU time (compared with the COBOL compiler, or even EDIT/3000). Because QEDIT does its own internal blocking and deblocking of records, it can reduce the CPU time used in the ile system by opening files with NOBUF/MR access.

II. D. QEDIT and "Constant Demands"

Most QEDIT commands are so fast that they are over before a serious strain has been placed on the host machine. For example, a 2000-line source program can be searched for a string in four seconds. For those operations that still are too much load, QEDIT provides the ability to switch priority subqueues dynamically. In fact, the system manager can dictate a maximum priority for compiles and other operations that cause heavy system load.

II. E. QEDIT and "Common Events"

The design of QEDIT is based on the fact that program editing is not completely random. When a programmer changes line 250, he is more likely to require access to lines 245 through 265 next, than to lines 670 through 710. This observation dictated the design of the indexing scheme for the QEDIT workfile. There are many examples of optimizing for the most common events in QEDIT:

- Each block of a QEDIT workfile holds a "screenful" of lines, with leading and trailing blanks eliminated.
- QEDIT has built-in commands to compile, PREP and RUN (since these functions are frequently used by programmers).
- QEDIT has a fast /SET RENUM command (it can renumber 600 lines per second), instead of a slow GATHER command.
- QEDIT can TEXT a workfile much faster than a KEEP file (since most text will end up in QEDIT workfiles).
- QEDIT can "undo" the DELETE command (because programmers are always deleting the wrong lines).

II. F. Results of Applying the Principles to QEDIT

In less than seven seconds, QEDIT can text 1000 lines, renumber them, and search for a string. Commands are 80% to 1200% faster than EDIT/3000, program size is cut in half, and disc I/O and CPU time are reduced by up to 90%. There are now more than 350 computers with QEDIT installed, in all parts of the world. Recently, we asked the QEDIT users what they would tell another user about QEDIT. Here are some of their answers:

> "If he's doing program development, he needs QEDIT." (Gerald Lewis, Applied Analysis, Inc.)
>
> "Would not live without it. $INCLUDEs in FORTRAN; one file or dataset per include-file." (Larry Simonsen, Valtek, Inc.)
>
> "Fantastic product." (Lewis Patterson, Birmingham-Southern College)
>
> "Buy it. The productivity advantages are tremendous and don't cost anything in machine load. The disc savings in a large (13 programmers) shop will pay for it." (Jim Dowling, Bose Corp.)
>
> "It's great. We usually get into QEDIT and just stay there for a whole session. Compiles and PREPs are very easy. I really like FIND, LIST, and BEFORE commands. QEDIT is very fast. It is great for programmers." (Larry Van Sickle, Cole & Van Sickle)
>
> "It's a tremendous tool and should be used by any medium-sized shop. I use it to produce an index of all source or job streams for an account." (Vaughn Daines, Deseret Mutual Benefit Assoc.)
>
> "QEDIT is the best editor I've used on the market. It makes a programmer extremely efficient and productive. In rewriting an existing system completely, the on-line compile, flexible commands, and savings of disc space all contributed to bringing the system up very rapidly." (Glenn Yokoshima, HP Corvallis)
>
> "Excellent product. Increases programmer productivity dramatically (morale too!)." (David T. Black, The John Henry Company)
>
> "FAST, convenient. No need to TEXT and KEEP. Somewhat dangerous for novice, because changes are made directly. [It worked well for us in] conversion of SPSS, BMDP, and other statistical packages to the

HP3000." (Khursh Ahmed, McMaster University)

"If you are writing a lot of programs, you should get QEDIT. It is much easier than EDITOR for this purpose. Program source files demand complex editing capabilities, which QEDIT has. I shudder to think of having to work on a 4000-statement SPL source using EDITOR rather than QEDIT." (Bud Beamguard, Merchandising Methods)

"Excellent product. Anyone using the HP editor more than 6 times per day (or more than 1 hour/day average) should not be without QEDIT!" (T. Larson, N. J. McAllister and Associates Ltd.)

"Easier to use than HP editor and much more efficient. I do not have to leave QEDIT to RUN, PREP." (Myron Murray, Northwest Nazarene College)

"Takes a great load off the mind (i.e., the "electronic brain"). There have been occasions when heavy editing would have killed our system if we had been using EDITOR." (Mike Millard, Okanagan Helicopters Ltd.)

"Very good product — works well in development environment. Compilation of source programs without leaving QEDIT is very nice for debugging." (David Edmunds, Quasar Systems Ltd.)

"Use it. It is so much better than HP editor that there is no comparison." (Ilmar Laasi, TXL Corp.)

"Fast text editor." (F. X. O'Sullivan, Foot-Joy, Inc.)

"In one word. Fantastic." (Tracy Koop, Systech, Inc.)

"Superb tool. Far better than EDIT/3000. Also, information about HP3000 that is supplied gratis is very useful." (James McDaniel, The UCS Group Ltd.)

"I would highly recommend it over EDIT/3000. In benchmarks and actual use, it has proven to be much less load on the computer. In a University environment, we have many students and faculty editing programs at one time. QEDIT allows us to run with a high session limit and still get decent batch turnaround." (Dan Abts, University of Wisconsin — La Crosse)

"QEDIT is an excellent product for the price, and is one of the easiest ways to increase programmer productivity. The LIST command has been invaluable for cross-referencing data items in COBOL source programs." (Mark Miller, Diversified Computer Systems of Colorado)

"Absolutely. QEDIT has allowed us to control the development of systems (requiring off-line compiles, audit trails for source modifications) while actually increasing programmer productivity." (Jean Robinson, Leaseway Information Systems, Inc.)

"Get it! It's great. Cheap at twice the price." (Willian Taylor, Aviation Power Supply, Inc.)

"QEDIT is THE ONLY text editor that you should use in a development environment." (Craig T. Hall, Info-tronic Systems, Inc.)

"Much better than HP's editor, well supported, well documented and continually improving. An excellent product. We activate QEDIT from our job file generator and activate SPOOK from QEDIT for editing and testing output and job streams." (Patrick Hurley, Port of Vancouver)

"Excellent — can do more than Editor, faster, and saves disc space. In searching for a specific literal, QEDIT finds them all in one command [e.g., LIST "literal"]." (Larry Penrod, Datafax Computer Services Ltd.)

"We could probably not operate if QEDIT were not available." (Winston Kriger, Houston Instruments)

"Buy it, or another computer (a second HP3000, of course)" (John Beckett, Southern Missionary College)

"Best software package I've bought for our shop." (James Runde, Furman University)

SECTION III
HOW TO INCREASE BATCH THROUGHPUT

By a "batch job" I mean a large, high-volume, long-running task, such as a month-end payroll or financial report. Why is there any problem with this type of task? Because the batch job is only a poor, neglected cousin of the on-line session. "On-line" is "with it," new, Silicon Valley, exciting; "batch" is old, ordinary, IBM, and boring. The best people and most of the development resources have been dedicated to improving the on-line attributes of the HP3000. The result is predictable: batch jobs are beginning to clog many HP3000 processors. The overnight jobs are not completing overnight and the month-end jobs seem never to complete.

The methods for maximizing the throughput of a single batch job are not the same as for maximizing the response time of a large number of on-line users. The biggest difference: for an on-line application, it is seldom economical to optimize CPU usage. There isn't enough repetition to amount to much CPU time. But, a batch process may repeat a given section of code 100,000 or a million times. CPU time matters.

I have identified five general principles for increasing batch throughput. Not surprisingly, they differ significantly from the principles used to improve on-line response time:

- Bypass Inefficient Code (CPU hogs).
- Transfer More Information Per Disc Access.
- Increase Program Size to Save Disc Accesses.

- Remove Structure to Save Unneeded Disc Accesses.
- Add Structure for Frequent Events.

For each optimizing principle, there are three different tactics you can apply, with three levels of complexity and cost:

- Changes in the Data Storage (simplest and cheapest, since no programming changes are needed).
- Simple Coding Changes (still inexpensive, since these are "mechanical" changes which do not require re-thinking of the entire application).
- Changes to the Application Logic (the most complex and expensive, since the entire application may have to be re-designed).

III. A. Bypass Inefficient Code (CPU hogs)

Elimination of inefficient code is the simplest way to produce big throughput improvements, assuming that you can find any code to eliminate that is inefficient (or more general-purpose than needed).

For a number of reasons, IMAGE is usually more efficient than KSAM as a data management method. If you don't need "indexed sequential" as your primary access method, convert from KSAM files to IMAGE datasets. Or, if you don't need "keyed" access to the data, convert all the way from a data management subsystem to an MPE flat file, and use sequential searches. The more powerful the data access method, the more CPU time is required to maintain it.

Bypassing inefficient code is simply a matter of recoding parts of programs to substitute an efficient alternative for an existing method that is known to have poor performance. For example, the MPE file system is CPU-bound when handling buffered files, so converting to NOBUF access will save considerable CPU time (you transfer blocks and handle your own records). In IMAGE, use the "*" or "@" field list instead of a list of field names. In COBOL, re-compile your COBOL68 programs with the COBOL-II compiler and they will run faster. The FORTRAN formatter is a notorious "CPU hog"; either bypass it completely or learn its secrets. The third-party software tool, APG/3000 (application profile generator), should be helpful in identifying the portions of an application where the CPU time is spent (APG was written by Kim Leeper of Wick Hill Associates). Once APG has identified the key section of code, you might want to recode it in SPL/3000 for maximum efficiency.

As is usually the case, the biggest improvements are obtained by re-evaluating the logic of the application. For example, you should periodically check the distribution of all reports to see if anyone is reading them. If not, don't run the job at all — that is an infinite performance gain.

III. B. Transfer More Information Per Disc Access

Besides CPU time, the other major limit on throughput is the access speed of the discs. One way to transfer more information per disc access is to build files with larger blocksizes. The "block" is the unit of physical transfer for the file. A larger blocksize means that you move more records per revolution of the disc. However, there is a trade-off: increased buffer space and impact on other users. In on-line applications, you usually want a small blocksize. Below, I will explain NOBUF/MR access, which is a technique that allows you to "have your cake and eat it, too!"

Another way to transfer more useful information per disc access is to ensure that the data is organized so the records that are usually required together are in the same disc block. Rick Bergquist's DBLOADNG program (contributed library) reports on the internal efficiency of IMAGE datasets. For example, if it shows that the work orders for a given part are randomly dispersed throughout a detail dataset (necessitating numerous disc accesses), you can ensure that they will be stored contiguously by doing a DBUNLOAD/DBLOAD (assuming that part number is the primary path into work orders). For master datasets, DBLOADNG shows you how often you can find a specific entry with only a single disc read (the ideal). If DBLOADNG shows that multiple disc reads are often needed for a certain dataset, you may be able to correct the situation by increasing the capacity of the dataset to a larger prime number or by changing the data type and/or internal structure of the key field.

Don't overlook the obvious either. If you can compress the size of an entry by using a more efficient data type (Z10 converted to J2 saves six bytes per field), you can pack more entries into each block and thus reduce the number of disc accesses to retrieve a specific entry.

You can often increase the "average information value" of each disc access by re-thinking your application. For example, suppose you must store transactions in a database in order to provide some daily reports, many monthly reports, a year-end report, and an occasional historical report covering several years. If you store all transactions in a single dataset, the daily jobs will probably take three hours to find, sort, and total 100 transactions. Why not put today's transactions in a separate dataset and transfer them to the monthly dataset after the daily jobs are run? When the monthly reports are completed, you can move the data to a yearly dataset, and so on. This is called "isolating data by frequency of access." The fewer records you have to search to find the ones you want, the more information you are retrieving per access.

It is theoretically possible to transfer more information per second by reducing the average time per disc access. Typically, you attempt to improve the "head locality" (i.e., keep the moving "heads" of each disc drive in the vicinity of the data that you will need next).

Although it is hard to prove, it does seem that using device classes to keep spooling on a different drive from databases, for example, does improve batch throughput. Under MPE IV, you can also spread "virtual memory" among several discs. The next "logical step" is to place masters and details on separate drives. However, in all tests that I have run with actual datasets and actual programs, there was no consistent difference in performance between having the datasets on the same drive or on different drives. The dynamics of disc accessing on the HP3000 are very complex. Unless you have the time to do a RELOAD afterwards, don't move files around; the moving process itself (:STORE and :RESTORE) may fragment the disc space and eliminate the potential benefit of spreading the files. Remember Green's Law: "The disc heads are never where you think they are."

You can also improve overall batch throughput by recovering wasted disc accesses. The disc drives revolve at a fixed speed, whether you access them or not. Any disc revolution that does not transfer useful data is wasted. Multiprogramming attempts to use these wasted accesses by maintaining a queue of waiting tasks. Unfortunately, maximum throughput under MPE III coincided with JOB LIMIT = ONE (no multiprogramming!). Under MPE IV, however, I have obtained a 25% decrease in elapsed time on the Series III by running two or three jobs concurrently. Try it.

III. C. Increase Program Size to Save Disc Accesses

In on-line optimizing, we are always trying to reduce the size of the program (code, data, and CPU usage), so as to allow the system to provide good response time to more users at once. In batch optimizing, we do not want better response time (we won't be running 36 batch jobs at a time, so we don't have to worry about mix); we want better throughput. Since most of the on-line tricks actually make the program slightly slower, we should avoid them. Batch tricks usually consist of trading off a larger program size for a faster elapsed time.

You can often save disc accesses by storing data in larger "chunks," keeping more data in memory at any time. Larger blocks will accomplish this, as will extra buffers. MPE file buffers can be increased above the default of two via :FILE, but doing so actually appears to degrade throughput. KSAM key-block buffers are increased via :FILE (:FILE xx;DEV=,,yy :MNS where xx is the KSAM data file and yy is the number of key-block buffers), which will help for empty files (KSAM cannot deduce how many buffers it will need unless the B-tree already exists). IMAGE buffers are increased via the BUFFSPECS command of DBUTIL; this can be effective for a stand-alone batch job, but only if it works with a large number of blocks concurrently (i.e., puts and deletes to complex datasets with many paths).

Pierre Senant of COGELOG (the developer of ASK/3000) has an ingenious method for "increasing program size" dramatically. He has implemented "memory files." An entire file is copied in main memory and kept there. For a small file that is frequently accessed (e.g., a master dataset containing only a few edit codes that must be applied to many transactions), Pierre's method should save enormous numbers of disc accesses.

NOBUF access to files was mentioned above as a way to save CPU time. If you use NOBUF with MR access, you can save disc accesses also, but at the cost of a larger data stack. MR stands for "multi-record," and gives you the ability to transfer multiple blocks per access, instead of just one block. With a large enough buffer, you will reduce the number of disc accesses dramatically.

Since multi-block access is faster only if each block is an exact multiple of 128 words in length, you should always select a recordsize and blockfactor such that the resulting blocksize (recordsize times blockfactor) is evenly divisible by 128 words. The resulting blocksize need not be large; it need only be a multiple of 128 (i.e., 256, 384, 512, . . .). As I promised earlier, here is your way to have the best of both worlds. Build your files with 512-word blocks (i.e., 4 times 128, 8 times 64, 16 times 32) for on-line use, and redefine the blocksize to 8192 words in batch programs via NOBUF/MR access.

For a "stand-alone batch" job, you may as well set MAXDATA to 30,000 words. This allows sorts to complete with maximum speed and provides other opportunities for optimization. With a larger stack you can keep small master datasets in the stack (e.g., a table of transaction codes). When you have exhausted the 30,000 words of your data stack, there are always extra data segments, which can be thought of as "fast, small files."

Re-evaluate your view of the data. Databases are usually set up to make life easy for the on-line user (rightly). Their organization may not be optimum for batch processing. In order to provide numerous enquiry paths, a single word order may be scattered in pieces among seven different datasets, and may require up to 20 calls to DBFIND and DBGET for assembly. In a batch job, if you are going to have to re-assemble the same order many times, it may be more efficient to define a huge, temporary record for the entire order, assemble it once, and write it to a temporary file. Then you can sort the temporary-file record numbers in numerous ways, and retrieve an entire order with a single disc read whenever you need it. Of course, this wastes disc space (temporarily) and increases your program size.

III. D. Remove Structure to Save Unneeded Disc Accesses

"Structure" for data means organization, lack of randomness, and the ability to quickly find selected groups of records. It takes work to maintain a "structure," and the more structure there is, the more work (CPU time and disc accesses) it takes.

Study your data structures critically. Can you reduce the number of keys in a record? A serial search may be the fastest way to get the data. Can you eliminate a sorted path? Overall, the application may be faster if you sort each chain in the stack after reading it from the dataset (Ken Lessey's SKIPPER package has this capability), but only if you don't use the COBOL SORT verb.

Another type of "structure" is consistency. IMAGE is a robust data management system because it writes all dirty data blocks back to the disc before terminating each intrinsic call. You can make IMAGE faster, but less robust, if you call DBCONTROL to defer disc writes (only after a backup). Another IMAGE idea: don't use DBDELETE during production batch jobs. Just flag deleted records with DBUPDATE and DBDELETE them later, when no one is waiting for any reports. When you can, use a DBUPDATE in place of DBDELETE and DBPUT.

For KSAM, if you are planning to sort the records after you retrieve them, use "chronological access" (FREADC) instead of default access (FREAD). Default KSAM access is via the primary key; KSAM must jump all over the disc to get the records for you in this sorted order, just so you can re-sort them in another order! Also for KSAM, try to keep only one key (no alternate keys), do not allow duplicates (much more complex), and avoid changing key values of records.

I am grateful to Alfredo Rego for pointing out a useful way to "eliminate structure" from IMAGE. When you are loading a large master dataset, use a Mode-8 DBGET prior to the DBPUT in order to find out if the new entry will be a primary entry or a secondary entry. Load only primaries on the first pass, then go back and load the secondaries on a second pass. This effectively turns off the IMAGE mechanism known as "migrating secondaries," which although essential, is time-consuming when filling an entire dataset.

III. E. Add Structure for Frequent Events

I saved this for last because it is one of the most powerful ideas. Batch tasks usually repeat certain key steps numerous times. Batch tasks have patterns of repetition in them. If you make that key step faster by adding structure to it, or re-structure the application so that "like-steps" are handled together, you can make the whole task faster. Extra structure (code complexity or data complexity) is justified in the most frequent operations of batch processing.

Check your data structures for patterns that you could capitalize on. For example, if you have a file of transactions to edit and post to the data base, could the task be made faster if the file were sorted by transaction type (only do validation of the transaction type when it changes) or by customer number (only validate the customer number against the database when it changes)?

Here are more examples of adding structure. If you sort by the primary key before loading a KSAM file, you can often cut the overall time in half. When erasing an IMAGE detail dataset, sort the record numbers by the key field that has the longest average chain length and delete the records in that order. When loading a detail dataset with long sorted chains, first sort by the key field and the sort field. In all of these examples, throughput is increased by adding code structure to match the structure of the data.

If you frequently require partial-key searches on IMAGE records, use an auxiliary KSAM file (or a sorted flat file and a binary search) to give you "indexed-sequential" access, rather than only serial access, to your IMAGE dataset. (Mark Trasko's IMSAM product enhances IMAGE by adding an indexed-sequential access method to the other access methods of IMAGE.)

If you have used many IMAGE calls to find a specific record, remember its record number. Then, when you need to update it, you can retrieve it quickly with a Mode 4 DBGET (directed read), instead of doing the expensive search all over again. If certain totals must be recalculated each month, why not re-design the database so that they are saved until needed again? If something takes work to calculate, check whether you will need it again.

The general principle is: look for patterns of repetition and add structure to match those patterns.

SECTION IV.
BATCH OPTIMIZING EXAMPLE: SUPRTOOL

SUPRTOOL is a utility program for the HP3000 that was developed by Robelle Consulting Ltd. The objectives of SUPRTOOL are to provide a single, consistent, fast tool for doing sequential tasks, whether in production batch processing, file maintenance, or ad hoc debugging. Example tasks that SUPRTOOL can handle are: copying files, extracting selected records from IMAGE datasets (and MPE files and KSAM files), sorting records that have been extracted, deleting records, and loading records into IMAGE datasets and KSAM files. SUPRTOOL can't do everything yet, but we are adding new capabilities to it regularly (the most recent enhancements are a LIST command to do formatted record dumps and an EXTRACT command to select fields from within records). SUPRTOOL embodies many of the batch optimizing ideas discussed in the previous section of this document.

IV. A. SUPRTOOL and "Bypassing Inefficient Code"

By doing NOBUF deblocking of records, SUPRTOOL saves enough CPU time to reduce the elapsed time of serial operations visibly. For MPE files, NOBUF is now fairly commonplace (although it still isn't the default mode in FCOPY — SUPRTOOL is 6 to 34 times faster in copying ordinary files). Where SUPRTOOL goes beyond ordinary tools is in extending NOBUF access to KSAM files (a non-trivial task) and to IMAGE datasets (very carefully). By making only a few "large" calls to the FREAD intrinsic, instead of many "small" calls to DBGET (each of which must access two extra data segments, look up the dataset name in a hash table, re-check user access security, and then extract a single record), SUPRTOOL quickly cruises through even enormous datasets with only a minimal consumption of CPU time.

For example, here is a comparison of SUPRTOOL and QUERY, selecting records from a detail dataset containing 60,971 current entries which are spread throughout a capacity of 129,704 entries.

```
SUPRTOOL/Robelle
>BASE ACTIVE.DATA,5
>GET LNITEM
>IF ORD-QTY>10000
>XEQ
IN=60971. OUT=14479.
CPU-SEC=56. WALL-SEC=133.
```

```
QUERY/3000
>DEFINE
DATA-BASE =>>ACTIVE.DATA
>FIND LNITEM.ORD-QTY>10000
USING SERIAL READ
14479 ENTRIES QUALIFIED
(CPU-SEC=520. WALL-SEC=763.)
```

Notice that SUPRTOOL used 1/9th as much CPU time and 1/6th as much elapsed time. And, the QUERY FIND command only builds a file of record numbers; to print the 14,479 records, QUERY must retrieve each one from the dataset again. SUPRTOOL creates an output disc file containing the actual record images, not the record numbers. With suitable prompting, SUPRTOOL can do this task even faster (see below for the BUFFER command).

IV. B. SUPRTOOL and "Transferring More Information"

SUPRTOOL transfers more information per disc access by doing multi-block transfers between the disc and the data stack in main memory. If records are 32 words long and stored as four per block (for a blocksize of 128 words), reading multiple blocks can make a big difference. For 20,000 records, one block at a time requires 5000 disc accesses. Using a 4096-word buffer and reading 32 blocks at a time reduces the number of disc accesses to 157!

SUPRTOOL has an option (SET STAT,ON) that prints detailed statistics after each task, so that you can see how it was done and where the processing time was spent. For example, suppose you want a formatted dump in octal and ASCII of all the records from the file described above for the order "228878SU." Below are the commands and times for SUPRTOOL and FCOPY:

```
FCOPY/3000
>FROM=SUMMRY;TO=*SUPRLIST;SUBSET="228878SU",1;OCTAL;CHAR
EOF FOUND IN FROMFILE AFTER RECORD 19999
3 RECORDS PROCESSED *** 0 ERRORS
(CPU-SEC=78. WALL-SEC=114.)

SUPRTOOL/Robelle
>SET STAT,ON
>DEFINE A,1,8
>IN SUMMRY
>LIST
>IF A="228878SU"
XEQ
IN=20000. OUT=3. CPU-SEC=11. WALL-SEC=16.

     ** OVERALL TIMING **
CPU milliseconds:             10854
Elapsed milliseconds:         16254
     ** INPUT **
Input buffer (wds):           4096
Input record len (wds):       32
Input logical dev:            12
Input FREAD calls:            157
Input time (ms):              6304
Input records/block:          4
Input blocks/buffer:          32
```

Notice that SUPRTOOL was using its default buffer size of 4096 words. FCOPY had to make 5000 disc transfers, while SUPRTOOL only had to make 157. That is one of the reasons why SUPRTOOL finished in 1/7th the time and used 1/7th the CPU time.

IV. C. SUPRTOOL and "Increasing Program Size"

SUPRTOOL gets a great deal of its performance edge by doing its own deblocking: allocating a large buffer within its data stack, reading directly from the disc into the buffer, and extracting the records from the blocks manually. SUPRTOOL trades a larger program size for a faster elapsed time. But you don't need to stop with the 4096-word buffer that SUPRTOOL normally allocates. Using the BUFFER command, you can instruct SUPRTOOL to work with buffers of up to 14,336 words and observe the results with SET STAT,ON. Here is the same selective file-dump that took 16 seconds with 4096-word buffers, done with 8192-word buffers:

```
SUPRTOOL/Robelle
>BUFFER 8192
>IN SUMMRY
>LIST
>IF A="228878SU"
>XEQ
CPU-SEC=10. WALL-SEC=13.  [An additional savings of 3 seconds]
```

By combining SUPRTOOL with IMAGE, you can have small data blocks for on-line access and large data blocks for batch sequential access. Here is the same database extract as done above (in the QUERY vs. SUPRTOOL test). Instead of using 4096-word buffers, we will increase the buffer space to 14,336 words:

```
SUPRTOOL/Robelle
>BUFFER 14336
>BASE ACTIVE.DATA,5
>GET LNITEM
>IF ORD-QTY>10000
>XEQ
IN=60971. OUT=14479. CPU-SEC=46. WALL-SEC=104. [Saved 29 sec.]
```

IV. D. SUPRTOOL and "Removing Structure"

SUPRTOOL can optimize batch operations by "removing structure." NOBUF deblocking of MPE files and IMAGE datasets provides faster serial access by saving CPU time and reading larger chunks of data, but NOBUF deblocking of KSAM files does that and more: it also eliminates structure. When you read a KSAM file serially by default, the KSAM data management system does not return the records to you in "physical" sequence; it returns them to you "structured" by the primary key value, and this takes work — a lot of work.

KSAM must search through the primary B-tree to find the sequence of the key values, and must then retrieve the specific blocks that contain each records. Quite often, logically adjacent records may not be physically adjacent; in the worst case, each logical record requires at least one physical block read. The SUPRTOOL NOBUF access to KSAM files cuts through all of this and returns the raw records to you in physical order; the savings in time can be impressive and, if you are planning to sort the records anyway, there is no loss of function. SUPRTOOL only removes the structure that you were not going to use.

Another example of removing structure in SUPRTOOL is the SET DEFER,ON command. When used in conjunction with the PUT or DELETE commands, the DEFER option causes SUPRTOOL to put IMAGE into output-deferred mode (via a call to DBCONTROL). Normally, IMAGE maintains a consistent and robust "structure" in the database after every intrinsic call. If you are planning to make a large number of database changes and can afford to store the database to tape first, you may be able to cut the elapsed time in half (or more) by leaving the physical database in an inconsistent state after intrinsic calls. (DBCONTROL makes the database consistent again when you are done.)

Here is an example use of SUPRTOOL to find all work orders that are completed (status="X") and old (dated prior to June 1st, 1982), delete them from the dataset, sort them by customer number and work-order number, and write them to a new disc file. SET DEFER,ON is used to make the DELETE command faster:

```
SUPRTOOL/Robelle
>BASE FLOOR.DATA
>GET WORKORDER
>IF WO-STATUS="X" AND WO-DATE<820601
>DELETE
>SORT CUSTOMER-NUM;SORT WORKORDER-NUM
>OUTPUT WO8206
>SET DEFER,ON
>XEQ
```

Another way to look at SUPRTOOL is as follows: if a serial search is fast enough, you may not need to have an official IMAGE "path" in order to retrieve the records you need. On the Series III, SUPRTOOL selects records at a rate of two seconds per 1000 sectors of data.

IV. E. SUPRTOOL and "Adding Structure"

SUPRTOOL can optimize batch tasks by "adding structure" to data. One way to add structure is to sort data. Experiments have shown that sorting records into key sequence can cut the time to load a large KSAM file in half. SUPRTOOL easily reorganizes existing KSAM files by extracting the good records, sorting them by the primary key field, erasing the KSAM file, and writing the sorted records back into it — all in one pass.

You can also add "structure" to raw data by defining a record structure for it (QUERY can access IMAGE entries because they have a structure defined by the schema). Normally, regular MPE files are not thought of as having the same kind of record structure as IMAGE datasets. Why is this so? Because you cannot access the fields of the file's records by name in tools such as FCOPY, even if the structure exists. In SUPRTOOL, you can.

If you use SUPRTOOL to archive old entries from IMAGE datasets to MPE disc or tape files, you can later do selective extracts, sorts, and formatted dumps on those MPE files, using exactly the same field names as you did when the entries were in the database. (In fact, you can even put selected records back into a temporary database with the same structure and run QUERY reports on them.) Here is how SUPRTOOL associates structure with raw MPE files:

```
SUPRTOOL/Robelle
>BASE FLOOR
>INPUT WO8206 = WORKORDER            [implied record structure!]
>IF CUSTOMER-NUM="Z85626"
>LIST
>XEQ
```

And, since SUPRTOOL has access to the IMAGE database that the entries originally came from, SUPRTOOL can still format the entries on the lineprinter with appropriate field names and data conversions (similar to REPORT ALL in QUERY).

IV. F. Results of Applying Batch Rules to SUPRTOOL

Just before completing this paper, we sent a questionnaire to the users of SUPRTOOL, asking them what they would tell other HP3000 sites about SUPRTOOL. Here are their replies:

"I always recommend SUPRTOOL with any new system. Without programming, I duplicated a master file from one application to another application. I set up a job stream to do this on a weekly basis (i.e., purge the old dataset entries and add the new dataset entries easily). SUPRTOOL creates files with different selection criteria to feed the same program." (Terry Warns, B P L Corp.)

"An essential package for efficient operation of a system. Most of our job streams include a SUPRTOOL function." (Vaughn Daines, Deseret Mutual Benefit Assoc.)

"Excellent. We had an application that serially dumped a dataset of 185,000 records (4 hours) and then sorted the 114-byte records in 6 hours (provided we had the disc space needed). We changed to SUPRTOOL with the OUTPUT NUM,KEY option and a modified program using DBGET mode 4 and maximum BUFFSPECS. The result was 4 hours altogether." (Bobby Borrameo, HP Japan)

"SUPRTOOL is an excellent utility for copying standard MPE files and databases very quickly . . . extracting and sorting records from a database (i.e., 40,000 records of 60,000), copying files across the DS line (much quicker than FCOPY), copying tape to disc and disc to tape." (Dave Bartlet, HP Canada)

"We couldn't operate without it. We are a heavy KSAM user and SUPRTOOL has cut our batch processing by at least 1/3." (Jim Bonner, MacMillan Bloedel Alabama)

"All sorts of marvelous things. [SUPRTOOL] is really nice (and fast) to copy a database for test pruposes or to make minor changes (instead of DBUNLOAD/LOAD) — even major changes, using a program to reformat the SUPRTOOL-created file." (Susan Healy, Mitchell Bros. Truck Lines)

"Just last night I told a friend that, after working with different sorts on IBM (DPD- and GSD-level machines), Burroughs sorts, and even HP sorts, SUPRTOOL is the best sort tool I have ever used." (Robert Apgood, Whitney-Fidalgo Seafoods)

"Get it. Runs much faster than SORT. Cheap at twice the cost." (Willian Taylor, Aviation Power Supply, Inc.)

"Fast and functional. SUPRTOOL is deeply embedded in our applications, most extracts are done with SUPRTOOL. Ad hoc inquiries [via SUPRTOOL], involving pattern matching on our customer file, extract the appropriate keys, which are then passed to the report program." (Patrick Hurley, Port of Vancouver)

"SUPRTOOL is a product which no shop that uses IMAGE and does batch report generation should be without. By changing certain reports to use SUPRTOOL instead of traditional selection techniques, a savings of 60% in CPU and wall time was obtained." (Vladimir Volokh, VSI/Aerospace Group)

"SUPRTOOL is a great timesaver when used with BASIC (or RPG) to modify IMAGE datasets and place them in another dataset or the same dataset." (John Denault, Datafax Computer Services, Inc.)

Thoughts Concerning How Secure Is Your System?

Ingenieurbüro Jörg Grössler
IJG, Berlin

WHAT DATA SECURITY MEANS

- To be able to rebuild the file system in case of a disaster
- To restrict access on various type of data.

STANDARD FILE BACKUP FACILITIES IN MPE

- Sysdump, Reload (based on magnetic tape)
- Store, Restore (tapes)
- User Logging (based on disc or tape)
- Private volumes (disc)

PROBLEMS WITH STANDARD FILE BACKUP

- Tape read error during RELOAD
  - — system cannot be started
  - — next action "must be RELOAD"

  measures:
  - — change disc packs before RELOAD
  - — RELOAD with "ACCOUNTS-only" then RESTORE the remaining files (very time consuming)
- Tape read error during RESTORE
  - — all files stored behind error point cannot be restored

  measure:
  - — use RESTORE or GETFILE2 program
- User logging causes system overhead

  measure:
  - — consider special logging during program design

PROSPECTS FOR TAPE-BACKUP SYSTEM

- GETFILE-facility will be improved
- Special STORE-RESTORE system is considered (this possibility includes features like UPDATE and APPEND)

RESTRICTIONS IN DATA ACCESS

- Account-system (users, groups, accounts with different passwords)
- User capabilities (SM, PM, PH, etc.)
- Filenames with passwords
- Privileged files
- File access capabilities on user/group- and file-level
- RELEASE/SECURE-commands

SEVEN POSSIBLE WAYS TO CRACK THE SYSTEM

1. FIELD.SUPPORT

   measure:
   Password on SUPPORT-account
   Or remove SUPPORT-account from the system.
2. Jobs in PUB.SYS-group

   measure:
   Password on job-file or
   Put job into other SYS-group.
3. LISTUSER@.@;LP

   measure:
   Log-on-UDC or perform command
   Not in PUB.SYS-group.
4. Open all files of the system

   measure:
   Special analysis of system logging
5. Read terminal buffers (PM-capability needed)

   measure:
   Remove PM-capability
6. Reading tapes

   measure:
   Keep track of all tape-transactions also using system logging
7. FOPEN on terminals

   measure: ??
8. . . .

Online Database: Design and Optimization

Robert B. Garvey
Witan Inc.
Kansas City, Missouri

CONTENTS

FOUNDATION

A system language, GOALS, will be introduced to render systems and components.

A general set of principles will be presented incorporating the components and structures inherent in a structured system. The use of these components in the system life cycle and as a documentation system will evolve.

A general system architecture will be presented and an approach to interactivity will be discussed.

The detailed use of callable programs in the 3000 environment will be discussed with emphasis on improvement of system performance.

I am going to assume that you are first time users of a 3000 that you want to write online database systems, that you do not have some of the more typical real world problems like a conversion from another machine and that you are going to use VPLUS and IMAGE. I don't care what language you use unless it is RPG in which case much of what I say will not be true.

GOALS: A System Language

GOALS was designed to meet the following criteria:

- Provide good documentation throughout the lifecycle
- Ease maintainence
- Expedite development
- Provide users early understanding of System functions and restraints
- Improve project management and reporting
- Reduce resources required
- Optimize System performance and quality

Many of the above criteria can be achieved through reasonable structuring of the system . However many of the structuring techniques that are now popular are simply more trouble than they are worth. Yourdon, Jackson and certainly IBM's HIPO involve more work involve more work in their maintainence than rewards merit. Warnier comes closest to being worthwhile but cannot be reasonably maintained in machine sensible form.

GOALS will be described as a methodology only because it does what the popular "Methodologies" tout, and much more. We do not feel that any of the methodologies should be considered ends in themselves and more sacred than the system at hand. Once the principles are learned and applied the implications should be obvious and the apparent need for a methodology forgotten.

Documentation

General Statement

The purpose of documentation is to assist in the analysis, design, program design, maintenence and operation of a system. To those ends software documentation must be flexible, easily modifiable, current and easy to read. Witan has developed a system of documentation called GOALS which uses simple text files associated through control numbers to meet the criteria listed above. The following sections describe the general features of the structural notation used in GOALS and the General system structure used in system projects.

GOALS is used throughout the life of a project. It is used:

1. To state requirements
2. Render flow and components in the analysis phase
3. To develop, test and render a general design
4. As a pseudo code or structured English for detail design
5. As a high level programming language
6. As a project network descriptor.

GOALS: Structural Notation

Formal structuring permits three primitive operations: Sequence, Repetition and Alternation. Structural Notation was developed to meet the criteria of formal systems in a generalized way and was guided by the assumption that systems must be rendered in a machine sensible form. GOALS relies upon text sequences and key words as its basis. Structural Notation is the basis of the syntax of GOALS.

Following are the representations of the primitive structures using flowcharts and GOALS. The word PROCESS is used to represent a step, a process or an item depending on the use of the notation at the time.

GOALS Primitive Structures

SEQUENCE

```
FLOW              ---------
               <    BEGIN    >
                  ---------
                      !
                !--------------!
                ! PROCESS 1    !
                !--------------!
                      !
                ----------------
                ! PROCESS 2    !
                ----------------
                      !
                ----------------
                ! PROCESS 3    !
                ----------------
                      !
                  ----------
               <     END      >
                  ----------

GOALS          1   PROCESS 1
               2   PROCESS 2
               3   PROCESS 3
```

ALTERNATION

```
FLOW

           ----------
        <    BEGIN     >
           ----------
               !
               *
             *   *
           *       *
        *    IF X     * --- true----->!--------------!
           *       *                  !  PROCESS 1   !---!
             *   *                    !--------------!   !
               *                                         !
               !                                         !
             false                                       !
               !                                         !
               *                                         !
             *   *                                       !
           *       *                                     !
        *    IF Y     * ---true------>!--------------!   !
           *       *                  !  PROCESS 2   !---!
             *   *                    !--------------!   !
               *                                         !
               !                                         !
             false                                       !
               !                                         !
               *                                         !
             *   *                                       !
           *       *                                     !
        *    IF Z     *---true------->!--------------!   !
           *       *                  !  PROCESS 3   !---!
             *   *                    !--------------!   !
               *                                         !
                                                         !
```

```
          !                                          !
        false                                        !
          !                                          !
          !<-----------------------------------------<-
          !
      ---------
     <   end     >
      ---------
```

```
GOALS        IF X IS TRUE
                PROCESS 1
             IF Y IS TRUE
                PROCESS 2
             IF Z IS TRUE
                PROCESS 3
             .
```

REPETITION

```
FLOW

                   ----------
                  <   BEGIN    >
                   ----------
                        !
                        *
                     *     *
                  *           *
<-false---<*        IF Y          *----true----
 !                *           *               !
 !                   *     *                  !
 !                      *                     !
 ------                 !              !---------------!
< END  >                !<----------!  PROCESS 1      !
 ------                                !---------------!

GOALS          WHILE  Y <<  IS TRUE   >>
                  PROCESS 1
                     PROCESS 1A
                     PROCESS 1B
                     PROCESS 1C
               !
```

The exclamation point is used to signify control in the WHILE loop. If the condition is met the control passes to the statement following the (!) on the same level. If the condition is met the control passes to the first statement following the condition. Processes 1A through 1C were added to show a simple subsequence.

Data Structuring

GOALS is also used to represent data structure. As with control structure there are three general structures which can be represented.

Data items listed line after line represent sequence:

1. item-1
2. item-2
3. item-3

Subsequences are represented as sequences on a level below the item of which they are are a part.

1. item-1
   1A. item-1A
   1B. item-1B
   1C. item-1C
2. item-2
3. item-3

Repetition in data structuring can be represented by "(S)" at the end of the item name which is repeated, this can take the form an expression [i.e., (0>s<15)].

item-1(S)
  item-1

Example: a file of accounts

  Account File
Account(s)
  Account
  Account number

```
Name
Address(s)
   Address type (h=home, w=work)
   Street number
   Direction
   Street name
   Affix
Amount due
Order(s)
   Order number
   Item(s)
      Item
```

Alternation

Alternation is represented with the IF control word or with the notation (1,0).

```
IF segment descriptive code = 1
      material
IF segment descriptive code = 2
      supply
```

This convention is seldom used because the WHILE handles most situations for the case of data structuring.

The other type of alternation is within a string of data items where the item can either exist or not exist. Another way of representing a non-required item.

1. item-1
2. item-2
3. item-3(1,0)

This says that items 1 and 2 must exits or are required and item 3 is optional.

Discussion

The highest level of repetition within a data structure is assumed to be the key to the file or at least the major sort sequence. If additional keys are required they can be represented with the word KEY [i.e., item-3 (KEY)] or an additional data structure can be presented to represent the structure represented when the KEY is used.

GOALS can be used to represent logical structures as well as the physical implementations. It is important that the required logical views of data be derrived and documented before any physical structures be planned. A goal in system design is to have a one to one relationship between the physical and the logical structures of the system. The coding complexity is reduced appreciably as wellas the maintainence activity. An additional byproduct is the ability to use Query or other general inquiry languages in a more straight forward fashion.

LEVELS: are represented graphically with the use of indentation. The first character in a line is considered to begin an "A" level subsequent levels are indented an additional three spaces each.

Successively lower levels (higher value characters and more deeply indented) represent subordinate processes. As will be seen in the general system structure the highest most levels are controled by increments of time; years, quarters, months, days, etc. while lower levels are controlled by events or conditions.

CONTROL NUMBERS: The control numbers used in GOALS are developed by alternating the use of numbers and letters to represent sucessively lower levels within the system. The system is similiar to English outlining except that only capital letters and numeric characters are used. For a given statement there is nothing to indicate its position in the hierarchy unless the entire control number is dipicted or the starting control number on the page is given. When GOALS statements are machine stored the entire control number is either stored or is assumed.

Principles

An Information system is distinguished from operating systems, command interperters, compilers and the like. An Information System is that set of communications, operations, files and outputs associated with a single conceptual "file."

I am not talking about a single program. Historically I am talking more about an application area.

Elements

Components

First an analogy: All purely mecanical devices are made up of elemental components; the incline plane, the wheel and axle, the lever and the chamber. The physics of these basic components and the materials from which they are constructed define the limits of their application. You may be saying, that list does not sound correct or "what about the screw." In listing elemental components certain definitions are inherent. I define the screw as a "rolled incline plane."

For information systems I assert that the list is: Communications, files, operations and outputs. The limits for such systems are defined by the ordering of the elements using the primitive structures (sequence, alternation and repetition).

As a note; to date the list of elemental components may have been input, process and output without regard to structure. This is more elemental considering all computer processes but is unbounded. This makes a general system design technique very difficult. Adding hierarchy to the above does not enhance these primitives to any great extent.

Relationships

With these boundries and definitions in hand, lets look at the relationships that develop.

There is generally a one to one relationship between file structure and operations structure, between communications structure and operations structure, between output structure and operations structure. In other words the operations or control structure mimics the other components of the system and each componet is related to the other in structure. Structure begins with the file structure.

Example; if you have a file of accounts and you want to report them; the report program may need to be structured exactly the same as the file or database to report all the data in the file. Most often there is a one to one relationship between files and outputs. In the report example the report structure could be expected to look exactly like the file. If the report is to look different than the file there would be in intervening operation usually a sort or selection to convert an intermediate output to the final output.

The same is true of communications which on the data processing level are the transmissions to the uses, the screens and the messages. The structure of a communication is generally the same as the operation structure which is the same as the data structure and thus the communication structure is the same as the data structure. This substantiates the theory that systems can be completly described knowing only the data structure. True but limited. Knowing the structure of any part should in theory give you the whole.

If everything describable about a system can be described in simple structures (and thus in GOALS) and the components of a system include only communications, files, operations, and outputs and GOALS can be used in all system phases then we have a framework for a general system covering conception through maintainence.

Lets look at any application. Traditionally you would begin with a requirements statement and do an analysis of the existing system. Forget flowcharts, classic narratives, and other charting techniques. Think of progressively decomposing the system using simple english outlining starting with the functions. Functions fit into the operations structure discussed. You will note that as you get down a level or two you will encounter repetitive tasks dependant on conditions, add WHILE and IF to your outlines and keep describing. Remember that users can understand outlines and repetition and alternation are not difficult to understand.

Operations will include existing machine processes, manual proceedures, paper flows, sorting processes etc. As you are going through the operations keep a list of the files that are mentioned and note the file keys (and sorts) and any advantages or requests for multiple keys.

List any outputs or reports prepared by the organization or required in the future.

Communications will be minimal at this stage but note any memos that may go from one section to another of a "file" of notes used as crossreference or duplicate of any more perminent file.

Your documentation is now shaping up; your notebook and I assume that the whole world has change to 8½"S11", should be divided into communications, file, operations and outputs.

The starting point for design is the detailing of the files in your file list. You will want to reduce the files as much as possible to a single file. By way of naming conventions the "file" should have the same name as the system at hand.

You will notice that many of the manual files are really communications in that they are "views" of the file that are required in a particular subfunction.

The design of the conceptual file must be validated against the required operations. I am going to leave this hanging for a moment to discuss a General System Structure.

General System Structure

A General System Structure is presented on the following page in Goals.

This structure is not applicable in all systems but is used as a pattern for system discription, design and understanding.

The key elements of design of this structure are:

1. File unity; a system with this structure has only one conceptual file. It may have any number of datasets of or physical files but they must be formalized into one.
2. Journalizing or logging; all changes made to data items can be (and normally should be) logged.
3. Last action dating; incorporated as part of logging, permits an offline log.

One detailed implication of this is need to have a date stamp in each detail set and a master date stamp in the master file.

Note: sleeper from the contributed library is a must. A standard job stream to prepare the system for shutdown and to bring it back up to production mode is also recommended. Allocation of application programs a desirable feature is the reason for this and also a good way to get sleeper going again.

```
                    General System Architecture

Begin system
WHILE NOT EOSystem
   WHILE NOT EOYear
       WHILE NOT EOQuarter
           WHILE NOT EOMonth
               WHILE NOT EODay
                   WHILE ONLINE
                       Begin online
                       identify operator and security
```

```
                      Open system file
                      Open current files
                      WHILE  Communication
                         IF control transfer
                            transfer control
                         IF batch request
                            initiate request
                         IF update , add or delete
                            Begin
                            Memo to LOG
                            LOCK
                            Update ,add or delete
                            UNLOCK
                            End
                         IF inquiry
                            perform communication operation
                         .
                      !
                   !
                !
                End Online
                Begin daily batch
                Perform daily batch processing
                Run LOG analysis
                If end of week
                   Perform Monthly Processing
                .
                ROLL FILES
             !
             perform Monthly processing
          !
          Perform Quarterly processing
       !
       Perform end of year processing
    !
    Close system
    End
```

A GENERAL DESIGN

With this Architecture and database design complete we have the basis for the development and implementation of any application.

Step 1 is inquiry into our file; if there is only one search criteria then we calculate into to file and return the master data or a summary. Once positioned in a master we can chain through our detail sets or follow appropriate programatic paths.

The master screen (a communication) should provide inquiry, update, and addition ability.

Each detail set should have a screen providing the same update add and inquiry ability. Our screens will be one for one with the detail sets. Think of a detail set as having a buffer that will correspond to communication (VPLUS) buffer. Moving data within one program is facilitated with this concept.

The list of detail sets becomes a list of programs which must be written to handle the retrieval, update, addition, deletion and editing of data for the detail set.

When this is complete you will have a functioning system; it will not function well. I have intentionally oversimplified. The office proceedures which may be in place or will evolve will dictate what combination of sets will appear on a screen but no effort was be lost in developing the barebones system according to this method. Each set (detail set) should have its own program to handle retrieval and update. When requirements demand inclusion the programs can usually be used with few changes. You can take this one step further to include a general scheme to handle multiple data sets on one screen.

The question then becomes; "How do I tie this all together?"

Interactivity and Control

Let's say that we have written a system composed of a series of programs that correspond to our data sets. The way in which we implement interactivity is through a control program called MENU. 4A Menus

A master data set will exist at the top of the conceptual file and the primary search path will be the file key. Other search paths will be provided through subsystems such as "Name Family" or through automatic masters. For all detail sets associated to the master there will be a program to handle that data set. Your analysis will dictate all the processes that the operator may wish to perform.

As other requirements develop associating more than one data set the code can be combined and new screens developed.

The menu control program provides transfer of control. It can do this either "quietly" or "loud." Loud is the obvious implementation; the operator choses a data set from a menu screen, the control is transferred via a "call" to a dynamic subprogram the data set is accessed updated, etc. and control returns to the controlling menu. But let us give the operator the ability to "tell" the system where he wants to go next. If he does a common area flag can be set to say don't display the menu simply transfer control to some other subprogram. We call are common area for data SYSBLK and out flag(s) Q1, Q2, etc. (you are not limited to one level of menu).

A menu structure may look like this:

```
      MAIN MENU
  WHILE NOT PARENT OR END OF SYSTEM
     IF LOUD
        GET MAIN MENU SCREEN
        SEND (SHOW) SCREEN
        WHILE  EDITS'FAILED
           EDIT FIELD
           IF EDITS FAIL
              SEND SCREEN
           .
        SET MODE TO QUIET
     IF QUIET
        IF NEXTPROCEEDURE=A
           CALL A
        IF NEXTPROCEEDURE=B
           CALL B
           ...
        IF NEXTPROCEEDURE =N
           CALL N
        ELSE
           CALL CONTROL'NUMBER'TABLE
        .
     !
```

Through this technique those programs which are not being used are not using memory resources. The CONTROL NUMBER TABLE refers to implementations which have levels of menus. If the control reference is not handled at that menu level control is appropriately passed to the proper level where a control program can handle it.

The quiet "CALL" technique can be used for any of the data set programs discussed by putting the quiet call structure "around" the program and requiring the passing of appropriate data into or from the communication buffer. If you need to pass data from one subprogram to another and you want to release the calling program stack space you can do so with extra data segments (DMOVIN, DMOVOUT) or message files or scroll files (logical device dependant files) that you set up in the application program i.e. BUILD SCROL033;rec=-80,16,f,ascii.

Pitch for the use of intrinsics; we have found that most 3000 users do not take advantage of some of the very rich intrinsics in MPE. They are simple calls, well documented and even those that require bit settings are fairly easy to implement in any language.

The COMMAND intrinsic, for example lets you issue MPE command line, commands programatically. We use this to create stream jobs then kick off the job from online programs. A report menu can be used this way.

Effect of called programs on the Stack

The effect of using properly implemented called programs is simple and dramatic. You reduce the amount of stack (that normally translates into main memory) that is required by each user of an application program. Jim Kramer HP SE Saint Louis (Quad Editor Fame) calls it timesharing the stack.

Usually the outerblock program carves out the required amount of data area to be shared by all subprograms in the "system"; this would normally include a database area, a VPLUS area and an area for the system at hand. MPE then carves out some data area for Image and VPLUS. Using a simple menu concept as discussed, as each program is called it will require its own data area and thus addition stack on top of the common (Q relative) data area, when the program returns to the menu this stack space will be unused but as soon as the next program is called this same space will be used by that program for its space.

COBOL sections do not do the same thing. They create data areas for all declared data in the data division. Sectioning permits smaller code segmentation but this is a shared resource on the 3000 anyway. Note that with stack sharing per user that the reduction in memory requirements is greatly enhanced over code optimization.

You will also find that editing code is much easier with smaller source files, that compilation is faster and more concise code is written.

SL's and USL's

SL's

- Modules, entry points and called Programs require 1 CST entries if they are not already referenced in a running process.
- Code is sharable by all programs. The PUB.SYS SL is avalable to all programs. Account and group SL's are available to programs being run out of that Account or group.
- You may exclusive access to the SL to make an entry in it.
- When SL entries are made you do not need to prepare the SL. It is available after you have exited the segmenter.

USL's

- Programs compiled into a USL must be prepared before they are runnable.
- Many programs may be compiled into the same USL. When a program is run the system will look to the USL for resolution of called programs, it then looks to the PUB.SYS SL unless a library is specified in the RUN. (RUN prog;LIB=G)

- All USL resolved entries create XCST entries except the outer block.

CST's and XCST's

- There are 192 CST entries available to user processes
- There are 1028 XCST entries available to user processes.

COMPILE INTO A USL COBOL/3000 Example

```
!JOB JOBNAME,username/userpass.accountname/accountpass;OUTCLASS=
!COBOL progname,$NEWPASS,$NULL
!SEGMENTER
USL $OLDPASS                    **  only needed   **
NEWSEG progname,progname'       **      for       **
PURGERBM SEGMENT,progname'      **  COBOL/3000    **
USL yourusl
PURGERBM SEGMENT,progname
AUXUSL $OLDPASS
COPY SEGMENT,progname
EXIT
!TELL user.acct; yourprog ---> yourusl
!EOJ
```

PREP OF USL

```
!JOB DyourUSL,user/userpass.account/accountpass;PRI=ES;OUTCLASS=
!PURGE yourrun
!CONTINUE
!BUILD yourrun;DISC=2500,1,1;CODE=PROG
!SEGMENTER
USL yourusl
PREPARE yourrun;MAXDATA=16000;CAP=MR,DS
EXIT
!TELL user.acct;  yourrun ---> yourrun
!EOJ
```

CALLABLES INTO SL's

```
!JOB D!SL,user/userpass.account/accountpass;OUTCLASS=,1
!COBOL yourprog,$OLDPASS,$NULL
!SEGMENTER
AUXUSL $OLDPASS
SL SL
ADDSL yourprog
EXIT
!TELL user.acct;  yourrun ---> yourrun
!EOJ
```

MENU

```
REPEAT until parent or end of system
   IF loud
      get menu screen
      show screen
      REPEAT until edits pass
         edit fields
         IF edit fail
             send screen
      !  .
      set mode to quiet
      .
      IF quiet
         IF nextprocedure = "O"
            CALL "O" USING ., ., .
         IF nextprocedure = "I"
            CALL "I" USING ., ., .
               .
               .
               .
         IF nextprocedure = "n"
            CALL "n" using ., ., .
         ELSE
            CALL "CONTROLNUMBERTABLE" using nextprocedure
            .
   !  .
```

Goals-SPL Standards

Section	Title
1	General
2	Procedures and Declarations
3	Moves
4	IF Control
5	REPEAT Control
6	Witan include files
7	Coding rules

GOALS-SPL STANDARDS

General

Indentation of three spaces indicates the beginning of a new level. If the next line is indented six spaces it indicates a continuation of the previous line.

Assignment is done with the ":=."

Comparison is done with the "=".

The astrisk is used to indicate that the address required in a statement has already been loaded on the stack. This has general applicability but we will limit its use to moves where the previous move has used the stack decrement option leaving the ending address on TOS. In a MOVE WHILE there is a stack decrement feature, a ",1" following the A, AN or N indicates that the final destination address is left on TOS.

The asterisk in parenthesis (*) indicates a backreference to another data item causing a redefinition of the area in the data stack. This back reference does allocate one word of the stack as a pointer.

Parameters should always be on word boundries thus BYTE ARRAYS should not be used as parameters.

Procedures and Declarations

Procedures parameters should all be called by reference not by value.

The form for an outer block program is:

```
$CONTROL USLINIT [ ERRORS=5, LIST, ... ]
BEGIN   << SOURCE >>
   [global data declarations]
   [procedures/intrinsics]
   [global-subroutines]
   [main-body]
END. <<  SOURCE  >>
```

The form for a subprogram is:

```
$CONTROL SUBPROGRAM [ERRORS=5, LIST, ... ]
BEGIN  <<  SOURCE  >>
  [compile time constructs]
  [procedures/intrinsics]
END.   <<  SOURCE  >>
```

The form of a sample subprogram using the Witan INCLUDE files found in the appendix follows:

```
$CONTROL SUBPROGRAM,ERRORS=5,NOLIST,NOWARN,SEGMENT=SEGNAM
BEGIN << SOURCE >>
$INCLUDE INC1G.T

<< BEGIN EXTERNAL PROCEDURE DECLARATIONS >>
$INCLUDE STDINTR.T << STANDARD EXTERNAL PROCEDURE DECLARATIONS >
PROCEDURE BLANK(WINDOW,VI);
   VALUE VI;
   IA WINDOW;
   IN VI;
   OPTION EXTERNAL;
<< END EXTERNAL PROCEDURE DECLARATIONS >>

PROCEDURE SEGNAM(VBLK,SYSBLK,RTN'CDE);
   IA VBLK,SYSBLK;
   IN RTN'CDE;
BEGIN << SEGNAM >>

<< BEGIN DATA >>
$INCLUDE VBLK.T
$INCLUDE SYSBLK.T
IA IBLK(0:0);
$INCLUDE SUBGLOB.T     << USING SUBGLOB.T REQUIRES THAT VBLK,IBLK
                          SYSBLK HAVE BEEN INCLUDED IN THIS PROCE
                          EITHER AS PASSED PARAMETERS OR AS NULL
                          ARRAYS. >>
<< OTHER DATA LOCAL TO PROCEDURE >>
LG KEEP'GOIN;
IN VI;
IN MISC;
IA (0:9)TEN'WORDS;
<< END DATA >>

<< BEGIN SUBROUTINES >>
SUBROUTINE PUT'WINDOW;
   BEGIN << PUT'WINDOW >>
      V'PUT'PAUSE(VBLK,2);
      BLANK(WINDOW,30);
      WINDOW'LEN:=60;
      VPUTWINDOW(VBLK,WINDOW'LINE,WINDOW'LEN);
      VSHOWFORM(VBLK);
   END; << PUT'WINDOW >>
<< END SUBROUTINES >>

<<************************************>>
BEGIN  << CODE >>

KEEP'GOIN:=TRUE;
WHILE        KEEP'GOIN DB
   KEEP'GOIN:=FALSE;
END'REP;

END; << CODE >>
END; << SEGNAM >>
END; << SOURCE >>
.
```

Moves

General Forms:

MOVE destination:=source, (length)[,stack decrement];

Literals:

Length need not be specified in the move of a literal If successive moves are anticipated to build a string or concatenate into a buffer then the stack decrement option of 2 can be used. Example:

MOVE OUTBUF:= "Hello",2
MOVE *;=" Everyone";

Non-Literals:

SPL requires type compatibility in moves, therefore general buffers should be defined in words and in bytes. The word buffer name should end with "'W." The byte buffers will have the just name without an identifying sufix.

The length parameter in the move should specify a name equated to the length in bytes or words depending on the type of move. The equate will generally be generated by DBUF. Byte lengths will begin with "BL' ", word lengths with "WL'."

Example:

MOVE OUTBUF:=
ACCOUNTNO,(BL'ACCOUNTNO);

Some moves may embed procedure to insure type compatibility and at the same time perform the appropriate conversion.

IF Control

The control structure for the IF will follow directly the structure enforced in GOALS. All IF's will be followed by a condition which may be compound and may extend to subsequent lines (note; continuation line disciple in general standards).

Following an "IF" condition a TB will be inserted, which is defined as a "THEN BEGIN." SPL does not require a BEGIN if the following statements are not compound, i.e., a lone statement. However, the "BEGIN" is required to bracket the sequence and to enforce the use of an "END" on the same level as the beginning "IF." If there are subsequent "IF's" on the same level (mutually exclusive IF's — programmer enforced) the IF should be converted to an IF'G which is defined in INC1G as an "END ELSE IF." This is not called a "IF" in GOALS. It is refered to as an "IF string" (mutually exclusive conditions).

Nested IF's:

If "IF's" are nested, the nested IF may begin any time after the "TB" of the preceding IF and will be indented to show its nesting. The rules for the nested IF are exactly the same as the IF; TB required.

ELSE

When the trailing ELSE is required in an IF string, the preceding end for the IF must not have a semicolon. The ELSE requires a BEGIN-END pair to enforce the terminating "END" at the end of the IF string.

Nested IF strings, where trailing elses come together may cause some confusion, but do not require any special rules.

Example:

```
IF --condition--          TB   << THEN BEGIN >>
  IF --condition--        TB
     --statm't--;
     --statm't--;
  IF'G --condition--        TB
     --statm't--;
     --statm't--;
  ELSE'G
     --statm't--
  END'IF;
ELSE'G
  --statm't--;
  --statm't--;
END'IF;
```

Repeat

General Form:

```
WHILE        --condition--          DB
   --statm't--------)
   --statm't---------
END'REP;
```

The REPEAT in the GOALS-SPL is used as documentation and is defined as a null statment. REPEAT must be followed by WHILE and a condition or compound condition. Following a WHILE condition a "DB" is required which is DEFINED in INC1G as a "DO BEGIN." As in the IF construct a "BEGIN" is required to enforce a terminating "END'REP."

```
BYTE POINTER
   BP  << USED FOR TEMPORARY POINTER, NOT SAVED >>
  ;
EQUATE
   RTN = 13  << CARRIAGE RETURN IN ASCCI >>
  ,ESC = 27  << ESCAPE CHARACTER IN ASCII >>
  ;
INTEGER I,J,K,LEN80,OLD'LANGUAGE;

  DA IBLK'D        (*) =  IBLK;
  BA IBLK'B        (*) =  IBLK;

  DA SYSBLK'D      (*) =  SYSBLK;
  BA SYSBLK'B      (*) =  SYSBLK;

  DA VBLK'D        (*) =  VBLK;
  BA VBLK'B        (*) =  VBLK;
DEFINE
   EL = END ELSE#
  ,END'IF = END#
  ,END'REP = END#
  ;
```

INC1G.T

This INCLUDE is used for abbreviation of data types and some constructs for GOALS presentation SPL compilations.

```
DEFINE <<USED TO ABBREVIATE DATA TYPES>>
   IA = INTEGER ARRAY#
  ,IN = INTEGER#
  ,DI = DOUBLE #
  ,LA = LOGICAL ARRAY#
  ,DA = DOUBLE  ARRAY#
  ,BA = BYTE    ARRAY#
  ,RA = REAL    ARRAY#
  ,XA = LONG    ARRAY#
  ,LP = LOGICAL PROCEDURE#
  ,DB = DO BEGIN#
  ,TB = THEN BEGIN#
  ,LG = LOGICAL#
  ,REPEAT = #
  ,G'IF = END ELSE IF#
  ,G'ELSE = END ELSE BEGIN#
  ,IF'G = END ELSE IF#
  ,ELSE'G = END ELSE BEGIN#
  ;
```

IBLK.T

```
<< IA IBLK(0:42); >>
   DEFINE
      COND'WORD       = IBLK #,             MODE4         = IBLK(29) #,
      STAT2           = IBLK(1) #,          MODE5         = IBLK(30) #,
      STAT3'4         = IBLK'D(1) #,        MODE6         = IBLK(31) #,
      STAT5'6         = IBLK'D(2) #,        MODE7         = IBLK(32) #,
      STAT7'8         = IBLK'D(3) #,        MODE8         = IBLK(33) #,
      STAT9'10        = IBLK'D(4) #,        ALL'ITEMS     = IBLK(34) #,
      BASE            = IBLK(10) #,         PREV'LIST     = IBLK(35) #,
      MODE1           = IBLK(26) #,         NULL'LIST     = IBLK(36) #,
      MODE2           = IBLK(27) #,         DUM'ARG       = IBLK(37) #,
      MODE3           = IBLK(28) #,         NUM'BASE      = IBLK(38) #,
                                            IBLK'LEN      = 43 #
                                            ;
```

IBLKG.T

The following is initilization code to be included in the outer block program to set IBLK fields :

```
MODE1              := 1;
MODE2              := 2;
MODE3              := 3;
MODE4              := 4;
MODE5              := 5;
MODE6              := 6;
MODE7              := 7;
MOVE ALL'ITEMS     := "@;";
MOVE PREV'LIST     := "*;";
MOVE NULL'LIST     := "0;";
DUM'ARG            := 0;
```

VBLK.T

```
<<  THIS ASSUMES THAT VBLK IS DECLARED IA VBLK(0:51)           >
<<     VBLK IS MADE UP OF COMAREA AND THE OLD VBLK             >
<<  CALLS TO VIEW INTRINSICS WILL USE VBLK AS THE COMAREA PARM >

   <<SPL DECLARATIONS FOR COMAREA>>
   DEFINE
      COM'STATUS          = VBLK (0)   #,
      COM'LANGUAGE        = VBLK (1)   #,
      COM'COMAREALEN      = VBLK (2)   #,
      COM'USRBUFLEN       = VBLK (3)   #,
      COM'CMODE           = VBLK (4)   #,
      COM'LASTKEY         = VBLK (5)   #,
      COM'NUMERRS         = VBLK (6)   #,
      COM'WINDOWENH       = VBLK (7)   #,
      COM'LABELSOK        = VBLK (9)   #,
      COM'CFNAME          = VBLK'B (10*2)   #,
      COM'NFNAME          = VBLK'B (18*2)   #,
      COM'REPEATAPP       = VBLK (26) #,
      COM'REPEATOPT       = VBLK (26) #,
      COM'FREEZAPP        = VBLK (27) #,
      COM'CFNUMLINES      = VBLK (28) #,
      COM'DBUFLEN         = VBLK (29) #,
      COM'DELETEFLAG      = VBLK (32) #,
      COM'SHOWCONTROL     = VBLK (33) #,
      COM'PRINTFILNUM     = VBLK (35) #,
      COM'FILERRNUM       = VBLK (36) #,
      COM'ERRFILNUM       = VBLK (37) #,
      COM'FM'STORE'SIZE   = VBLK (39) #,
      COM'NUMRECS         = VBLK'D (21)     #,
      COM'RECNUM          = VBLK'D (22)     #,
      COM'TERMFILENUM     = VBLK (48) #,
      COM'TERMMODE        = VBLK (49) #,
      COM'TERMALLOC       = VBLK (50) #,
      COM'DATAOVERRUN     = VBLK (51) #,
      COM'READTIMEOUT     = VBLK (52) #,
      COM'OTHERDATAERR    = VBLK (53) #,
      COM'MAXRETRIES      = VBLK (54) #,
      COM'TERMCONTROLOPT= VBLK (55) #,
      COM'TERMOPTIONS     = VBLK (55) #,
      COM'ENVINFO         = VBLK (56) #,
      COM'TIMEOUT         = VBLK (57) #
   ;
   EQUATE
      COMAREALEN          = 60,
      COBOL'LANG          = 0,
      VBLKLEN             = 100,
      SPL'LANG            = 3,
      MAXWINDOWLEN        = 150,
      MAXMODELEN          = 8,
      NAMELEN             = 15,
      NORM                = 0,
      NOREPEAT            = 0,
        V'REPEAT            = 1,
        REPEATAPP           = 2,
        ENTERKEY            = 0,
        PARENTKEY           = 1,
        KEY2                = 2,
        KEY3                = 3,
        REFRESH             = 4,
        PREV                = 5,
        NEXTKEY             = 6,
        INQ'ENT             = 7,
        EXITKEY             = 8
     ;
     <<SPL DEFINITIONS FOR VBLK>>
     DEFINE
        WINDOW'LEN          = VBLK  (COMAREALEN+0)      #,
        MODE'LEN            = VBLK  (COMAREALEN+1)      #,
        WINDOW'LINE         = VBLK  (COMAREALEN+2)      #,
        WINDOW'MODE         = VBLK  (COMAREALEN+2)      #,
        WINDOW              = VBLK  (COMAREALEN+MAXMODELEN)      #,
        WINDOW'LINE'B       = VBLK'B  ((COMAREALEN+2)*2) #,
        WINDOW'MODE'B       = VBLK'B  ((COMAREALEN+2)*2) #,
        WINDOW'B            = VBLK'B  ((COMAREALEN+MAXMODELEN)*2) #
     ;
```

SYSBLK.T

```
<<IA SYSBLK(0:114)  SPACE ALLOCATED IN MAIN PROGRAM >>
DEFINE
     CNTRL'NUM          = SYSBLK #,
     LST'PROC           = SYSBLK(2) #,
     NXT'PROC           = SYSBLK(4) #,
     Q1                 = SYSBLK(6) #,
     Q2                 = SYSBLK(7) #,
     Q'NEXT             = SYSBLK(8) #,
     OPER'ID            = SYSBLK(9) #,
     SECU'TY            = SYSBLK(11) #,
     SSC                = SYSBLK(13) #,
     CNUM               = SYSBLK(16) #,
     L'FLNUM            = SYSBLK(21) #,
     M'FLNUM            = SYSBLK(22) #,
     FLAGS              = SYSBLK(23) #,
     DQSTAT'SB          = SYSBLK(28) #,
     GLSTAT'SB          = SYSBLK(43) #,
     TERMID             = SYSBLK(58) #,
     MSBLK'SB           = SYSBLK(63)#   << STARTING ON DOUBLE BOUNDRY
     ;
DEFINE
     CNTRL'NUM'B        = SYSBLK'B #,
     LST'PROC'B         = SYSBLK'B(2*2) #,
     NXT'PROC'B         = SYSBLK'B(2*4) #,
     OPER'ID'B          = SYSBLK'B(2*9) #,
     SECU'TY'B          = SYSBLK'B(2*11) #,
     SSC'B              = SYSBLK'B(2*13) #,
     CNUM'B             = SYSBLK'B(2*16) #,
     FLAGS'B            = SYSBLK'B(2*23) #
     ;
EQUATE
     CNTRL'NUM'BL       = 4,
     LST'PROC'BL        = 4,
     NXT'PROC'BL        = 4,
     OPER'ID'BL         = 4,
     SECU'TY'BL         = 4,
     SSC'BL             = 6,
CNUM'BL          = 10,
FLAGS'BL         = 10
;
```

Coding Rules

All agorithms should first be done in GOALS without concern for the SPL structure. SPL constructs will be used for individual statements and conditions but the control structure should be in GOALS.

This complete:

1. Replace all ELSE's with G'ELSEs or ELSE'Gs.
2. Locate all "IF's that are on the same level as a "running" IF. Replace each running IF with an IF'G or G'IF.
3. Replace all ".'"s with an END'IF;
4. Insert a THEN BEGIN or "TB" following every IF condition.
5. Replace all "!" with an END'REP;.
6. Insert a "DB" or DO BEGIN after every REPEAT condition.

```
            An Example using the rules on the preceeding page

              WHILE         ------------
                 IF -----------
                    --------------
                    --------------
                    IF -----------
                       --------------
                    ELSE
                       --------------
                       --------------
                    .
                 IF -----------
                    --------------
                 .
                 IF -----------
                    --------------
                 IF -----------
                    --------------
                 ELSE
                    --------------
                 .
              !

                                                  <<   SPL      RULE >>
              WHILE         ------------     DB   << DO BEGIN      6  >>
                 IF -----------              TB   << THEN BEGIN    4  >>
                    --------------
                    --------------
                    IF -----------           TB   << THEN BEGIN    4  >>
                       --------------
                    ELSE'G                        << END ELSE BEGIN   1 >>
                       --------------
                       --------------
                    END'IF;                       << END'IF        2  >>
    ***(20) ERROR ***
   LINE
   1490
    TRUNCATED BY  4 CHARACTER(S)
                 IF'G ---------              TB.  << END ELSE 2 >>  << THEN B
                    --------------
                 END'IF;                          << END'IF        3  >>
                 IF -----------              TB   << THEN BEGIN    4  >>
                    --------------
    ***(20) ERROR ***
   LINE
   1495
    TRUNCATED BY  4 CHARACTER(S)
```

```
         IF'G ---------                TB    << END ELSE 2 >>   << THEN B
             --------------
         ELSE'G                              << END ELSE BEGIN  1 >>
             --------------
         END'IF;                             << END'IF        3  >>
     END'REP;                                << END'REP       5  >>
```

Note: work the top example yourself using the rules and see if it matches the completed program. Note the count of the begins and ends match for SPL. Do the algorithm correctly in GOALS and the SPL code will follow.

Auditing with IMAGE Transaction Logging

Robert M. Green
Robelle Consulting Ltd.
Aldergrove, B.C., Canada

SUMMARY

The transaction logging of IMAGE is not just for recovery of lost transactions; the transaction log-files contain a vast array of information that is useful for auditing purposes. Reports generated from these files can answer basic audit inquiries (WHO, WHEN, WHERE, WHAT and HOW), can provide statistics that are useful for performance tuning (which dataset has the most puts and deletes?), and can aid in program debugging (what does this program actually change in the M-CUST dataset?)

CONTENTS

INTRODUCTION

Since MPE release "1918," the IMAGE/3000 database system has had the ability to "log" database changes to a disc or tape file. Although the format of these log-files is somewhat obscure (and not documented accurately or completely), they can provide a great deal of information that is useful for auditing. DBAUDIT (a proprietary software product of Robelle Consulting Ltd.) will analyze transaction logging files and print transaction audit reports from them.

Here are two sample IMAGE transactions (a DBOPEN and a DBPUT), as printed by DBAUDIT, which show the auditing information that is available from transaction logging:

```
OPEN    17 AUG81 11:38 N:3          L:1
U:BOB.GREEN,PUB                P:QDBM.PUB.GREEN
B:TEST.PUB.GREEN             Mode:1   Security:64
Logon device:28   as a session.  Userid:None.
Last dbstore:17 AUG81 11:15

PUT     17 AUG81 11:38 N:4          L:1
U:BOB.GREEN,PUB                P:QDBM.PUB.GREEN
B:TEST.PUB.GREEN             DE:DCOM             R:10
Data-Added:
ORD-NUM          = 11111111
COM-NUM          =       5
COM-DESC         = 5555555555555555555555555555555555555555
```

Each transaction has a date and time stamp (17 AUG81 11:38), a unique transaction number assigned by MPE (N:3), a unique logging access number for each user who does a DBOPEN (L:1), a logon account, group and user name (U:BOB.GREEN,PUB), a program name (P:QDBM.PUB.GREEN), a base name (B:TEST.PUB.GREEN), a logon device number and batch/session indicator, an optional userid (an extra identifier that can be passed to DBOPEN as part of the password, and acts to distinguish between different users who happen to be logged on with the same MPE user name), the dataset type and name (DE:DCOM is a detail dataset), the data entry's physical record number (R:10), the key-field value (for master dataset entries), the fields that were added, deleted or changed (with field names), plus before and after data values that have been converted from binary to ASCII where necessary (ORD-NUM = 11111111).

Logging Answers Basic Questions

As you can see, logging provides answers to many questions:

```
WHO (logon user name and user id)
WHEN (date and time)
WHERE (database and dataset)
WHAT (data fields changed)
HOW (terminal number and program name)
```

The only question that cannot be answered is WHY?

Two Types of Log-files

There are two basic types of logging files that can contain IMAGE transactions: raw log-files (on disc or tape) which are filled by IMAGE as transactions occur, and user recovery files generated by DBRECOV during transaction recovery.

The original log files have fixed-length records, with large transactions split over several records.

The user recovery files hold the transactions as variable-length records (one record per transaction). User recovery files are usually generated for transactions that DBRECOV could not recovery (and which you must recover by hand). User recovery files contain one extra field which is not found in regular log-files: a recovery flag that indicates whether each transaction was successfully recovered or not ('OK' or 'NO').

SELECTING AND FORMATTING THE TRANSACTIONS

Since a great deal of paper can be consumed in printing every detail of every transaction, DBAUDIT has commands to restrict which transactions are selected and what data is printed for the selected transactions (if any).

The *SELECT* command allows you to select transactions for specified bases, datasets, programs, users, time periods, and a range of record numbers.

```
>SELECT BASE TEST
>SELECT DATASET TEST,DCOM
>SELECT USER BOB.GREEN
>SELECT BEFORE 1132
>SELECT NFROM 212
```

The *LIST* command allows you to control which transactions are printed (*LIST* CALLS) and which field values are printed for the transactions (*LIST* FIELDS). In order to print only the date and time of DBOPEN transactions, these commands would be used:

```
>LIST CALLS NONE
>LIST CALLS O+
>LIST FIELDS NONE
>LIST FIELDS D+T+
```

Here are the full options of *LIST:*

```
>LIST FIELDS N+D+T+C+L+U+P+B+S+R+K+F+M+X+
These flags determine how much information is
printed for each transaction:
L:  unique id number assigned by DBOPEN
U:  user.account,group name
P:  program.group.account
B:  basename.group.account
S:  set name/type; MA=master, DE=detail
R:  IMAGE record number of entry
D:  date of transaction
T:  time of transaction
N:  sequence number assigned by user logging
C:  call type (OPEN, CLOSE, ...)
F:  field values
K:  key-field value
M:  memos from DBMEMO, DBBEGIN, DBEND
X:  extra fields on DBOPEN (mode, etc.)

>LIST CALLS  O+C+P+D+U+B+E+A+M+T+L+
These flags enable/disable listing of the different
IMAGE and MPE intrinsic CALLS in the log-file.
  O=open C=close P=put D=delete U=update B=begin E=end
  A=abort of program between BEGIN and END calls
  T=termination of program without calling DBCLOSE
  L=logging status records (header, trailer, ...)
```

OTHER USEFUL INFORMATION

DBAUDIT also provides three other useful pieces of information for the auditor or database administrator:

1. Reliability of the log-files for recovery purposes.
   DBAUDIT checks each log-file to ensure that transactions are in the proper sequence (for date, time, transaction number and logging access number). If there are any inconsistencies in the log-file, they are detected and reported. Without DBAUDIT, the only way to test a log-file is to restore the original database and actually run a logging recovery. DBAUDIT's double-checking feature has already detected a number of bugs in IMAGE transaction logging.
2. Detection of program aborts.
   DBAUDIT reports program aborts separately from regular program DBCLOSEs. By selecting only the abnormal termination transactions, you can see which programs are aborting. This can be helpful in ensuring program quality.
3. Detection of program "abends."
   DBAUDIT reports programs which terminate with an unmatched DBBEGIN. This can happen because the program aborted during a logical transaction, because the programmer forgot to terminate the logical transaction with a DBEND, or because the system crashed during a logical transaction. DBAUDIT gives you a quick summary count of the number of ABENDs in the file (it should normally be zero), plus additional details if the count is non-zero (where in the file the ABENDs occur, whether the DBBEGIN has a put, delete, or update after it, etc.).

SUMMARY TOTALS AND STATISTICS

From this basic transaction data, it is possible to generate a number of useful summary statistics.

Transaction Breakdowns

One way of analyzing the transactions is to break them down into the different types of transactions, and then total them by program, user, base and dataset:

```
  For all bases, current:   1 entries  20 maximum.
TEST.PUB.GREEN
O:1         X:1         P:2         D:0         U:0
   DCOM                 P:2         D:0         U:0

  For all progs, current:   1 entries 200 maximum.
QDBM.PUB.GREEN
O:1         X:1         P:2         D:0         U:0

  For all users, current:   1 entries 200 maximum.
BOB.GREEN,PUB
O:1         X:1         P:2         D:0         U:0
```

In these tables, O equals DBOPENs, X equals DBCLOSEs, DBBEGINs, DBENDs, and DBMEMOs, P equals DBPUTs, D equals DBDELETEs, and U equals DBUPDATEs. Note that for datasets (such as DCOM), only P, D and U totals are collected.

Summary Totals

Here are the types of summary totals provided by DBAUDIT:

```
Logging Records Read from File              32
Transactions that were Selected             6
  Transactions read but not Selected        0
Transactions Selected and Printed           6
  Transactions Selected, not Printed        0
Transactions,but no OPEN (should be 0)      0
Inconsistencies in File (should be 0)       0
Number of ABENDs in file (should be 0)      0
```

FUTURE POSSIBILITIES

Information that could be generated from the log-files, but is not currently collected by DBAUDIT, is the total "changes" to a given numeric field, such as ACCOUNT-BALANCE in a CUSTOMER-MASTER entry. By periodically summing the field values in the entire database and comparing changes in this sum with the incremental changes to that field (as recorded in transaction logging), it should be possible to ensure that all transactions are being logged (i.e., the database is in balance with the transactions).

One problem is the database in which IMAGE schema does not accurately describe the actual data fields. This situation usually happens when users over-

k. Terminate the programs that are generating the transactions; when the last database accessor has closed the database, MPE will terminate the log process and close the logfile. Now, use DBUTIL to disable logging to this database (so that DBAUDIT can access it!).

```
:RUN DBUTIL.PUB.SYS
>DISABLE TEST FOR LOGGING
>EXIT
```

l. Run DBAUDIT and specify TESTLOG as the source of input records. If you do not want the report on the lineprinter, you can use a :FILE command to the file DBREPORT to redirect it to $STDLIST or to a disc file.

```
:FILE DBREPORT=$STDLIST;REC=-79
:RUN DBAUDIT.PUB.ROBELLE
>INPUT TESTLOG
>EXIT
```

5. Users report that the system load of transaction logging may not be as bad as was first rumored. Considering the performance improvement that is likely to accompany MPE IV, now may be a good time for users to consider activating transaction logging for their databases.
6. In one of the releases of MPE IV, IMAGE was changed in a way that can affect user programs. Previously, a user program could invoke the DBBEGIN, DBEND or DBMEMO intrinsics whether or not logging was active for the database in question. The intrinsics always returned STATUS=0, as long as the parameters were legal. Now, these three intrinsics return an error (non-zero STATUS) if logging is not active at the time of the call (i.e., the DBBEGIN, DBEND or DBMEMO could not be logged). User programs that don't care whether logging is running, must *not* check the STATUS result. User programs that wish to ensure that the operator has activated logging can now do so by checking the STATUS from these intrinsics.

Transaction Logging and Its Uses

Dennis Heidner
Boeing Aerospace Company
Seattle Washington

For some time database users have been concerned about the integrity of their databases and methods to prevent them from being corrupted. Another concern is performance measurement. When H-P introduced MIT-1918, they also introduced "Transaction Logging." Transaction logging is intended to provide a means of repairing databases which are either damaged or are suspected of being so. There are however many additional benefits to be derived from transaction logging including automatic audit trails, historical records of the database users, and information on the database performance.

The purpose of is paper to discuss the basic concepts of transaction logging, its benefits, and drawbacks. Various logging schemes, such as long logical blocks, and multiple IMAGE databases are discussed. Several different database logging cycles and HP recommended recovery procedures are discussed, and a method of recovering and synchronizing multiple databases is proposed.

Finally this paper covers a user written program which has been used to monitor the database performance, to validate and debug new user-written application software, and provide a complete audit trail for future reference.

INTRODUCTION

Many computers are justified only because they can keep track of large quantities of information in "real time" databases. In such cases it becomes extremely important that the integrity of this information remains consistent.

The database can be destroyed or corrupted in a number of ways. These include program errors, personnel errors, and computer hardware problems. A considerable amount of time and resources can be expended to eliminate most of the program errors, but it is almost impossible to guarantee a perfect program. The second source of inconsistencies is people. While it is possible to protect the information from human error by increasing the complexity of the program or by eliminating the human contact with the machine and its peripherals, both are often undesirable. Finally the third cause is system failures. System failures can be caused by numerous events including such things as fires, earthquakes, vandalism, hardware problems, power failures, and of course, MPE flaws.

We can take steps, however, to protect our investment in the database. There exist several very good programs,[1] such as DBCHECK and DBTEST, which will look for and can correct minor structural problems caused by crashes. But what about the user who must update a critical path in IMAGE? To do so requires a DBDELETE followed by a DBPUT. If the system crashes between the two, there will be no structural damage to be found. If you don't mind losing a $50,000 item or a $100,000 check, you have no worries. . . An effective database protection method is transaction logging. Logging takes many forms, the simplest of which only requires that we file away the paperwork used to generate the modifications to the database. Although this is convenient, it is a poor approach when it comes to recovering the database from a crash or system failure. For instance let's assume that we have a failure after two or three thousand transactions have been entered from terminals at several locations. Who wants to re-enter all the old data, while all the normal work is stacking up?

A better method is to have the computer keep duplicate copies of the information used to make the changes. Then it would only be necessary to instruct the computer to use the duplicate to reconstruct the database following a crash.

There are several ways that computers can be used to generate these duplicate copies. The most efficient method is to write the programs with an intrinsic transaction-logging system. This logging system can either be supplied by HP or could be a custom logging scheme. The problem with custom schemes is that they generally require as much or more design time as many of the applications programs that will use them. Since this work is not readily visible to either the end user or management, there is a temptation to do a quick job. The resultant lack of planning causes poor database and system performance. Additionally, in-house logging schemes only work with the in-house programs. If we use externally-written software (such as QUERY), we may find it difficult or impossible to get these routines to use our logging schemes.

TRANSACTION LOGGING (USERLOGGING)

HP recognized this need for database protection, and developed a version of transaction logging which runs on the HP3000.[2-3] HP's transaction logging is actually a process which runs under the control of the MPE operating system. If the database is enabled for logging, a logging process then attaches itself to the database when it is opened up for any update access. If the database is opened up in a read-only mode, the logging process is not attached. When the logging process is running it intercepts transactions after the IMAGE check has been made, yet before the actual transaction has been made in the database. This captured data (old and new values) are then blocked up in a buffer in memory. When the memory buffer fills up, the transactions are moved out to a logging file on the disc. If we are logging to the disc only, then this becomes our duplicate copy of the transactions. If we are logging to the tape drive, then the disc buffer is periodically moved out to the tape drive (see figure 1).[4]

If we have a system failure (or any other event which could cause a database inconsistency) then we use a database recovery procedure which uses a good copy of the database and the duplicate copy of the transactions to restore the information in the database to its condition only moments before the crash.

The recovery program which HP supplies is called DBRECOV. The program literally re-works all the transactions in the same sequence as originally made; this repetition assures that the database structure is correct and undamaged.

Once the database has been corrected and brought back into a consistent state, a backup copy is made and a new logging media is used. The act of making a backkup copy and using a new logging media is known as beginning the logging cycle.

In order to implement transaction logging, HP introduced several new user-callable DBMS procedures: DBBEGIN, DBEND, DBMEMO, WRITELOG, BEGINLOG, ENDLOG, OPENLOG, and CLOSELOG. These new procedures are extremely useful because they let us define how transactions are logically grouped.[5]

To illustrate the importance of logical grouping of transactions, assume we have two mutually-dependent pieces of information. It is important that if any change is made to one item, the change that is made to the second item must also be made. If either item is not changed, then neither should be modified. We can do this by using DBBEGIN to mark the beginning of the dependent changes, and DBEND to mark the end (see figure 2). The intrinsic routines ensure that if there is a system crash or failure between the DBBEGIN and the DBEND, neither transaction is made. While transaction logging does not guarantee that we will not have crashes, it does provide some relief in recovering from their effects.

Now let's talk about the drawbacks. Anytime we ask the CPU to perform additional work, there is an increase in the overhead cost for our process. The object is to balance the additional workload on the computer with the benefits that we hope to gain.

Every time the memory buffer is moved out to disc, or the disc buffer is moved to magnetic tape, these transfers tie up the disc controller. Although this may be for very short periods of time, one of the biggest problems plaguing many HP3000 sites is slow response time due to a large number of disc accesses.

If we install logging then, our response time may become worse. Your alternative of course is to use absolutely no logging at all! Thus transaction loggings may be one of the necessary evils in life.

LOGGING STRATEGIES

The placement of the calls to DBBEGIN and DBEND can play a crucial role in the success or failure of logging. Since each call to DBBEGIN or DBEND causes a logging record to be written, and thus additional overhead, it is tempting not to use these at all. The people in the logging laboratory at HP wrote DBRECOV to handle both blocked and unblocked transactions (QUERY does not block its transactions). However while this is ideal for existing programs, we may be losing some very valuable information about our databases.

By properly placing the DBBEGIN and DBEND it is possible to measure the performance of our database. This information can later be used to tune-up our applications programs. Additionally proper placement of the calls enhances our crash recovery procedures.

The worst possible thing that we can do is to take the easy way out, calling DBBEGIN when we open up the database and calling DBEND as we close the database. This results in large recovery blocks. As long as we never have a crash everything works fine. However the first time we must recover after a crash, we might find that DBRECOV is unable to help us out. This is because the recovery process tries to resolve all transactions made between periods when the database is inactive. With the long blocking scheme the database is almost always active. DBRECOV will attempt to build a monstrous file to look for dependent transactions, and inevitably fail!

HP recommends that we make all the necessary locks on the database, call DBBEGIN, make the transaction, the call DBEND before unlocking (see Figure 3A). This will ensure that we have a minimum chance of large concurrent blocks.[6]

Another strategy that appears to work well is to call DBBEGIN, then lock the database or sets, and make our updates. Conversely we would unlock, and then call DBEND (see Figure 3B). This method allows us to measure the time between the begin and the end, which reflects the performance of our database. This procedure works quite well, as long as the following conditions are met:

- Always use ASSIGN LOCK OPTION OFF in QUERY
- Our transactions are made by terminals, and designed so that they collect the data from the screen, perform edits, then go through the DBBEGIN, DBLOCK, updates, DBUNLOCK, DBEND.

If you cannot operate under these conditions, then stay with HP's recommendations.

MULTIPLE DATABASES

When HP first introduced transaction logging, they did not make any provisions for synchronizing transactions which span multiple databases. The DBBEGIN and DBEND intrinsics work only for a single database at a time.[7] However with MIT 2028, HP introduced the BEGINLOG and ENDLOG intrinsics. These new intrinsics now make it possible to develop a method for synchronizing multiple database transactions. This is done by calling BEGINLOG before any multiple database transaction, and ENDLOG at the completion of the transaction (see figure 4). A user-written program could then scan the transaction log for complete BEGINLOG-ENDLOG blocks and indentify the record number of the last complete transaction.

To recover the database you then run DBRECOV and specify @@CONTROL EOF=recordnum." It may be necessary to run DBRECOV for each database that was involved.

LOGGING CYCLES

The method and length of our logging cycles depends heavily on the application and previous experience with the computer system's reliability. There were several possible methods proposed by HP during the MPE 1918 update course. These include:

- DBSTORE, then start a new logfile
- DBSTORE, start a log tape, when it fills start a new one, when it fills start another . . ..
- SYSDUMP, start a logfile

The first logging cycle method is the perferred method. It is straightforward, the recovery procedure is easy to follow, and in the event of a system failure, downtime is limited to the time needed to recover one logfile.

The second type of logging cycle should only be used on databases which require backing up, but have very little activity. This is because each logfile complicates the recovery procedure, and adds a considerable amount of time to recover each logfile.

The third logging cycle option omits the DBSTORE. We have found that a DBSTORE takes about 2 minutes for 3 megabytes of database (1600 bpi tape, series 33 computer). At first glance it would appear that the use of DBSTORE wastes time. However DBSTORE sets some internal flags and time stamps which SYSDUMP does not. These internal stamps and flags are used by DBRECOV to provide added protection against using logfiles from the wrong time period.

If you use a SYSDUMP tape, you must remember to request SYSDUMP store all the files. If partial backups are done, the database must be restored from the latest full backup, then restored from each succeeding partial, before DBRECOV is used. Because the time stamp and flags were not set by SYSDUMP, we must then specify that DBRECOV is to ignore all time stamps and flags. This is often difficult or dangerous to do, especially if your system operators are inexperienced.

SYSDUMP should only be used as a backup for the previous two logging methods. If you do not want to have your database stored on your backup tapes, then you should look into Alfredo Rego's STORENOT program. STORENOT allows the creator of a database to "tie it up" so that it is not stored by full or partial backups.

The logfile can reside on either the disc or magnetic tape. It is faster to log to the disc; however, if the reason for the system failure is a disc hardware or free space problem, you could lose both your database and the backup copy of the transactions. The other choice is for the logfile to reside on tape. This has two drawbacks: first, it ties up the tape drive, and second, it periodically requires the CPU to move the logging buffer from the disc to tape. If the system is already heavily loaded this can only worsen the problem.

If you decide to log to a disc file, you should be careful to build the logfile large enough to hold all of your expected transactions plus a reserve. You can obtain a rough estimate of the log size by:

```
# of sectors = 4*number of database opens
               + (number of updates * up-
                 date rec len)
               + (number of puts * put re-
                 cord length)
               + (number of deletes * de-
                 lete rec len)
               + 1 for DBEND
               + 1 for DBBEGIN
      update rec len (in sectors)
         = (# of items in list
      + update buffer size)/256
      delete rec len (in sectors)
         = (# of items in list
      + delete buffer size)/256
    put record length (in sectors)
         = (# of items in list
         + put buffer size)/256
```

If the buffer sizes are not known — use the media record size . . . you can get that from a DBSCHEMA compilation.

You can count the # of items in the item list or if "@;" was used then just use the item count in that particular set.

If you are not sure you calculated the size correctly then use the :SHOWLOGSTATUS command to monitor the number of records in the log. If you run out of space in a disc file while logging, you can put the database in a state similar to a crash; this may require that you go through a complete database recovery procedure!

CRASH RECOVERY

HP implies that a recovery procedure must be followed every time there is a database crash.[8] This can be disastrous. On one occasion we followed the recommended crash recovery procedure, purged the database, restored the database, and started DBRECOV. It bombed, and upon investigation we discovered that approximately 500 transactions had been lost because the logtape was blank due to a tape drive malfunction. Moral of the story: You should first determine the cause of the crash, then verify that the logfile is good via LOGLIST or DBAUDIT.

We also found that it is important to write your applications programs so that they abort to prevent further transactions if they detect a logging problem. It is possible for the program to pass the IMAGE checks for DBDELETES, delete an item, then find out there is a logging problem! The end result is one less item in the database. This becomes especially critical if you are one of the many IMAGE users who have to update critical items by deleting and re-adding.

If the crash is because the logfile was too small and filled up, then the end result of trying to recover is that your data-entry personnel spend hours reconstructing previous transactions. It is better to run a program such as LOGLIST, and find out what data have been effected. Then run DBSTORE, build a new, larger logfile, and start a new logging cycle. One note of caution: we found that parity errors on the tape drive cause a crash whose symptoms are almost identical to those of one caused by running out of space on a disc logfile.

If the crash is because of a system failure, the correct procedure is:

- Perform a memory dump for HP
- WARMSTART (if possible); this causes MPE to try to recover the transactions in the internal disc buffer. (THIS IS VERY IMPORTANT!)
- SHUTDOWN
- COOL or COLDSTART
- Run LOGLIST or DBAUDIT to determine who, what, when and how bad the crash is.
- If the database was not open in an update or modify mode then simply start a new logging cycle and get your users back on.
- If the database was open in an update or modify mode, then purge the database using DBUTIL, restore the database using DBRESTOR and recover using DBRECOV. BE SURE TO START A NEW LOGGING CYCLE!

AUDIT TRAILS

Good data processing applications have some form of built-in controls which allow for the verification of the accuracy of the database. This is especially true if the application is in the banking, inventory control, or government fields. In many applications some form of an electronic "paper trail" is mandatory.

The information which is logged by IMAGE exceeds most audit requirements and can provide the required electronic trail. Transaction logging records information about who, when, where, and how an item or entry was modified. This information can be extracted in several ways. Bob Greene has a package called DBAUDIT which can analyze the log.[9] I have contributed a similar program called LOGLIST (via IUG 1982 swaptape) which can expand the transaction log per directions. It is described in a appendix to this paper.

The audit trail recorded by transaction logging can be enhanced by carefully planned use of the 'text' area on DBBEGIN, DBEND and DBMEMO. We record the information which leads to a transaction when we call DBBEGIN. The results of the update or special error conditions are logged on the DBEND. If needed, DBMEMO is used to record special remarks and initials of the person making the change.

If you foresee a requirement for frequent analysis of the transaction log, it is also important to include a time-stamp as an item in individual data entries. This forces IMAGE to log both the present time-stamp and its previous value. The value of this information is apparent when tracing the history of a specfic data entry. With a time-stamp on your data entries, it is possible to pull and analyze only those logfiles which contain the time interval about the time-stamp of interest. Since analysis of a transaction log takes about 10-15 minutes for 40,000 records, the time saved in this manner can be considerable.

Perhaps more importantly from a programmer's point of view, we can use the audit trail as a method of providing continuous software monitoring. The concensus among data-processing people is that it is virtually impossible to guarantee that a complex program will correctly handle all cases regardless of what data is fed to it. When an error does occur at our site, experience indicates that it is generally several months before we notice that something is wrong. By maintaining transaction logfiles for a sufficient length of time (6 months), it is possible to locate the source of most errors. This makes it much easier to correct latent program errors. In addition we have found that if the problem was caused by human error, the hard-copy printout that can be generated from the log tape goes a long way toward refreshing the memory of the person who made the mistake.

For users at sites whose software must be accepted by Quality Assurance, audit trails have an additional advantage. As part of the acceptance testing on new releases of our applications programs, we DBSTORE the database, then run the test programs and fully analyze the log. This enables us to provide a visual check on fields and items in a manner easier than using QUERY.

After using the transaction log as an audit trail and debugging aid during the last two years I would estimate that we have saved probably a hundred man-hours which would otherwise have been spent looking for the cause of "freak errors."

As with all good things in life there is a "Catch-22." IMAGE3000 is structured as a closely-knit group of files tied together with the root file. When modifications are made to the database , only the set number, item number and item buffer are logged. If the root file is altered (by using ADAGER, DBGROOM, etc.), then the link between the database and the transaction log is broken. The most obvious problem occurs when the order of data sets is changed with ADAGER's DETSLIDE. Suddenly your Employee-Detail becomes your Part-Master and the log analysis program either bombs or gives ridiculous answers. You have two choices: either don't use ADAGER (not a very realistic choice), or use ADAGER's SCHEMA to generate a dummy version of the database structure as it appeared before changes were made. Then use the editor to shrink the capacity of all the sets down to 3 or 5. Assign this schema some version number and identify on all logfiles under which version of the schema the logfile was made. I have set up a separate group in our account for these "old, shrunk databases." Then when I need to look at an old logfile, I set up a file equation referencing the old "database" and run LOGLIST under that condition.

TRANSACTION-LOGGING PERFORMANCE

There is a great emphasis on designing systems with better response time. For this reason any type of overhead (regardless of how beneficial) is generally shunned. To make matters worse, when HP introduced transaction logging with MIT 1918, they had indicated that there would be a "through-put reduction of 30% for large modication-intensive online applications running 10 or more concurrent processes."[10] Unfortunately the test environment used for that statement was not completely explained. During the past two years we have been using transaction logging on a Series 33 with 768 kbytes and typically 11 active processes. Our experience has shown that there was probably less than 10% reduction in throughput. So, what is the overhead cost of transaction logging?

In order to find out, I wrote a program (DBPERF) which allows me to benchmark IMAGE transactons with and without logging. The benchmarks are deliberately run with as light a load as possible in order to isolate the overhead caused by logging from the effects of other users' activities (see APPENDIX: DBPERF). The results of the tests are shown in Figures 5-7. In Figure 5 we see the comparison of the time to DBPUT verses pathcount, on series 33 and 44 CPU's. As seen in Figure 5 the added overhead caused by transaction logging, is marginal. The anomalies on series 44 data was caused by a user logging on and using FCOPY during the benchmark test. Figure 6 shows the comparison of the time to DBDELETE verses pathcount, on the series 33 and 44 CPU's. The overhead caused by transaction is marginal, again the anomalies on the series 44 data was caused by a user logging on and using FCOPY during the benchmark test. Figure 7 shows comparisons of the time to DBOPEN, DBUPDATE and block transactions with DBBEGIN and DBEND. Earlier I mentioned that logging blocks up the IMAGE transactions (approximately 32 transactions), then moves this buffer out to disc. The overhead caused by this movement is comparable to the roll-in and -out of an inactive user process by the memory manager (MAM).

In most on-line applications the overhead added to the transaction is considerably less than the threshold point at which the system becomes overloaded. However batch jobs are generally another story, if you have batch jobs which require a considerable amount of system resources, run them without logging. Store your database before the job begins, stream the job, and when it completes, then store the database and start a new logging cycle. If you have a crash during the unprotected batch jobs it will only require that you DBRESTOR and rerun the jobs.

PREDICTION OF RESPONSE TIMES

At this point it will be worthwhile to discuss a little queueing theory and how it is used to estimate response times so that we can illustrate the effects of transaction logging on the system. A queue is just a waiting line.[11] When we analyze queueing systems, we talk about such things as number of servers, arrival rate, transaction rate and number of users. The classical example of queues in operation is the waiting lines at banks. With only one cashier (number of servers), if the customers arrive at a rate of one per hour (arrival rate) and the cashier takes only 15 minutes to complete an average transaction (transaction rate), then there will be no waiting line and the cashier can perform some overhead functions such as washing windows while waiting for the next customer. If, on the other hand, customers arrive every 15 minutes, then we can expect to find a person at the cashier constantly. The windows start to collect dirt and grime since the cashier no longer has time to wash them. When the arrival rate of the customers increases to one every 10 minutes, we soon find that a line is forming. If sufficient time is allowed to pass, customers start to switch banks, the cashier demands a raise and the windows now appear to have several layers of dirt and grime and strange creatures crawling on them.

Transaction processing on an HP3000 performs in a similar manner. As long as the arrival rate is sufficiently slower than the transaction rate, MPE is able to perform its necessary overhead functions and the response time is good. Unfortunately the HP3000 cannot ignore its overhead as the cashier did, so as the arrival rate approaches the transaction rate, response time begins to suffer.

It is possible to estimate the response time of the computer if you are able to estimate the number of users, the average time each user "thinks" about what needs to be done, and the time required to complete the transaction. The average "think time" is equal to:

```
                  arrival rate
Think time =   ---------------
                number of users
```

For example:

The XYZ Company has an HP3000 Series 33 computer on which they wish to implement an application which will support 10 users. The "think time" of these users is about 30 seconds each per transaction. The transactions consist of a DBDELETE and a DBPUT on a detail set with four paths. What will their transaction response time be?

The transaction response time is equal to:

```
          Transaction response time= Queue length * transaction rate

          Queue length = the greater of
                                              1
                                             or
                         number of users * transaction rate
                         ----------------------------------
                                     think time

Using the IMAGE benchmark results, we then determine:

          Transaction response time = Queue length * 1.3 sec

                         10*1.3    13
          Queue length = ------  = --  ; as noted above, use 1
                           30      30

     then Transaction response time = 1.3 sec

Using XYZ adds logging, it will be:
```

```
If XYZ adds logging, it will be:

                         10*1.4    14
          Queue length = ------  = --  ; as noted above, use 1
                           30      30

     then Transaction response time = 1.4 sec
```

Our model works well as long as the computer has time to perform its overhead functions, i.e. code-segment swapping, MAM function, and garbage collection. The time available for the computer was approximately:

```
                        User think time
   Computer idle time = ---------------  - transaction rate
                        number of users
```

In the case of the XYZ company this averaged 1.6 seconds per user transaction (with logging).

The overhead that was added due to transaction logging is:

```
                        transaction   transaction
                        time with   - time without
                        logging       logging
       Added overhead = ----------------------------  * 100
                         transaction time without
                                 logging

    or, for XYZ,

                        (1.4-1.3)
       Added overhead = --------- * 100   =   7.6%
                           1.3
```

If 7.6% overhead is enough to cause XYZ's machine to have problems, can you imagine what an additional user using QUERY, the editor, or any of the compilers would do?

An additional benefit from transaction logging is that we are able to collect the arrival rates, transaction rates, and number of users during our actual production enviroment. With this knowledge we can make more accurate design decisions when developing new and additional applications.[12]

CRASH-PROOF?

How crash-proof is your database? Damage to databases can be caused in several ways. The typical cause of damage is a crash occurring while adding or deleting an item to or from a detail set. If the DBPUT or DBDELETE was manipulating the internal pointers in the database, then you can probably count on having at least one broken chain. Other types of database crashes occur when MPE or some "neat" privelege-mode program adds its own kind words to a random data set! When discovered, this error has the same symptoms as a broken chain; however, you may also be missing a considerable amount of data.

Perhaps the worst kind of database crash is the one you can't find. That is, DBTEST, DBCHECK, ADAGER and even DBUNLOAD-DBLOAD say everything is ok. These errors occur when the data set has a critical path which must be updated. Since IMAGE will not let us update critical paths, we have to delete and re-add. If a crash occurs after the DBDELETE is complete and before the DBPUT re-adds the item, then we have lost an entry in the database though the database structure remains intact (see Figure 8). DBTEST, DBCHECK and the other routines have no way of testing for or detecting this error. If your HP3000 is an accounting system, this is intolerable. This type of error could be prevented by using transaction logging and placing the DBBEGIN at the start of the transaction and DBEND at its finish.

It is possible to estimate your chances of having some form of damage to your database in the event of a crash. This Crash Figure of Merit (CFOM) is given by:

```
              (transaction rate * number of users)
CFOM = ------------------------------------ * 100
                         think time
```

If your CFOM is high, say 20 or 30 percent, then it is probably worth the effort to run DBTEST and DBCHECK on every database that was open when a crash occurred. It may also be very much worthwhile to try transaction logging. If the CFOM is very low (one to two percent), then it is probably easier to manually correct errors and run DBCHECK at some convenient time.

SUMMARY

This paper discusses the merits and drawbacks of transaction logging, and provides some basic guidelines to aid in the successful implementation of transaction logging. Since most applications are designed to "earn" money, it is only fair to treat transaction logging in the same manner. As summarized in figure 9, the decision to log or not to log should be made only after a careful review of the associated system costs, its performance cost, alternatives, and by establishing values for the intangibles such as improved data security, benefits from audit trails, etc.

ACKNOWLEDGEMENT

I wish to thank the HP sales office in Bellevue, Washington, for allowing me to run DBPERF on their Series 44.

APPENDIX LOGLIST

LOGLIST is a logfile analysis program written by the author; it has the following capabilities:

A. Show who, what, when, and how a database which was running with transaction logging was accessed.

B. Trace the changes made to the database and expand the values in a format similar to QUERY so that the dump is easily readable.

C. Selectively track user-requested database items which fall within user-specifiable limits.

D. Show when the log was opened, closed, or restarted and identify all users that were accessing the database during a crash!

E. Provide statistics showing the database activity, transaction elapsed time, detail sets accessed, the ratio of BEGIN-ENDS to database transactions, average transaction times, and worst-case transaction response time.

F. F. Identify (if any) the processes which had "broken" transactions.

Running LOGLIST

LOGLIST should be run in the same account and group in which the database resides. If the log to be examined is on disc, then that file must also be accessible. LOGLIST cannot analyze a logtape that is currently active. Finally, the log analysis consumes considerable CPU time (even though the elapsed time of the analysis may be very short). It is advisable the log analysis be either streamed in a low JOBPRIORITY (DS or ES) or run during periods of low computer usage.

LOGLIST Commands

LOGLIST commands are listed below, each followed by a short summary of its function.

HELP — print additional instructions

DATABASE=[dbname[.group[.acct]]]
(if not specified the values are set to @.@.@ and no expansion of the log records may be done. Only the Log User Summary and histograms will be generated.)

PROCESS=[program[.group[.acct]]]
(if not specified the values are set to @.@.@)

LOGON=[user[.group[.acct]]]
(if not specified the values are set to @.@.@)

LIST[=range]
expand the transactions made to the database (in the QUERY report format) showing:
the user that made the modification
if an UPDATE, what was changed
if a DELETE, what was deleted
if a PUT, what was added

The transactions are outlined in asterisks (*) to indicate indicate "logical transactions." When the beginning or end of a transaction cannot be determined, the program leaves the outlined block open (see Figure 10). On such blocks, the LOGID of the process is printed and it is possible to rerun the analysis — specifying that those items be expanded separately.

RANGE — The range field is optional, and is in the following form:
LIST=startingrecord:endingrecord

If the ending record is not supplied then LISTLOG will continue to expand until the end of the log file.

NOLIST disable expansion of the transactions made to the database

DATE=m1/d1/y1 [TO m2/d2/y2]
look only for transactions made between and including the specified dates. The default for m2/d2/y2 is 99/99/99.

TIME=H1:M1 [TO H2:M2]
look only for transactions made during the specified time interval. The default for H2:M2 is 24:00.

FIND dset.itemname (EQ,LT,GT[,IB])
'value1'[,'value2']
look only for transactions made to dset.itemname and falling within the bracketed area as specified by the relational operators.

FIND dset record#
look only for transactions made to record# of dset.

```
          { TAPE;LABEL=label            }
LOGFILE={                               }
          {filename[.group[.acct]]      }
```

if a filename is specified, you must have exclusive read-access to the file. If tape is specified, you must be able to use this non-sharable device.

RUN — begin processing the transaction log.

EXIT — exit the program and return to MPE.

SHOW — display current parameters.

INIT — initialize the files, plots and data back to the way they were when LOGLIST first started. Any data accumulated so far will be sent to the LP.

LIMIT — limit and identify the "worst" transactions. This causes all transaction response-time data which exceeds 20 times the current running average to be thrown out. The time of day, user and process are printed on $STDLIST. This command has no effect until ten logical transactions have been completed. It is useful in locating deadlocks.

<CONTROL Y> — ("CNTL" and "Y" keys pressed simultaneously) interrupt the program (sessions only). The program will give the the time and date of the transaction which it is currently processing and ask if you wish to continue. A "Y" or "N" is expected.

Interpretation of the Log User Summary (see Figure 11)

USER — Logon user name
GROUP — Logon user's group
ACCT — Logon user's account
DBASE — Database that was accessed
PROCESS — Process run by user
GROUP — Group in which the process resides
ACCT — Account to which the Process belongs
LOGON TIME — Time the process began
LOGOFF TIME — Time the process closed the database
LG# — LOGID # for the process (assigned by MPE)
DEV — Logical device from which the process was run
O — Database open mode
CAPABILITY — User's capability (see WHO intrinsic of MPE)
UP — Number of DBUPDATES
PUT — Number of DBPUTS
DEL — Number of DBDELETES
#BLKS — Number of complete logical transaction blocks

Inferences from the LOGLIST Statistics

Several histograms and charts are derived from the data; these are provided by LOGLIST to aid in the interpretation of the data.

DATABASE ACTIVITY (see Figure 12)

The DATABASE ACTIVITY histogram plots the number of transactions on the y-axis and the time of day (in 15 minute intervals) on the x-axis. This histogram can be useful in determining when the peak database loads occur.

DATABASE RESPONSE TIME (LOG10) (see Figure 13)

The LOG10 plot is a useful tool in determining if a process or processes are suffering from very bad response time or may be causing database deadlocks. The LOG10 plot covers the range from .1 sec to 10,000 seconds.

DATABASE RESPONSE TIME (LINEAR) (see Figure 14)

The LINEAR plot is a useful in determining if a process or processes are suffering from poor database response times. The y-axis represents the number of transactions made. The x-axis represents the time, from 0 to 30 seconds.

LOGICAL BLOCK SIZE (see Figure 15)

The LOGICAL BLOCK SIZE histogram is useful in evaluating the effectiveness of the transaction blocking of a process. This chart may also be used to determine if a program is calling the DBBEGIN-DBEND pair only at the beginning and end of processes or after making single database modifications.

DATABASE RESPONSE TIME (AVERAGE) (see Figure 16)

The AVERAGE histogram can be useful in evaluating modifications made to existing programs by aiding in the determination of whether or not the system (as seen by the database users) is getting slower or faster.

DATABASE RESPONSE TIME (WORST CASE) (see Figure 17)

The WORST CASE histogram is useful in locating processes that may have caused database deadlocks. The histogram is also useful in determining if there are certain times during the day in which stream jobs may be run with little or no impact on the response time for on-line users.

TRANSACTION FREQUENCY (see Figure 18)

The TRANSACTION FREQUENCY histogram is a measure of the time between logical blocks, often called the user's "think time." This plot, in conjuction with the database response time charts, can be helpful in determining if and/or how improvements can be made to the application programs and the system.

ADD-DELETE-UPDATE TO BEGIN-END RATIO (see Figure 19)

The ratio of DBPUTS, DBDELETES, and DBUPDATES to DBBEGINS and DBENDS is a good indication of how the transactions are blocked by the user's application programs. The desirable range is $0 \leq$ [PUTS + DELETES + UPDATES] / [BEGINS + ENDS] < 100.

If the ratio is less than one, this usually indicates that there is a process or processes which are making only one database transaction per BEGIN-END set. Although this is not harmful, it does not fully utilize the benefits of transaction logging, resulting in more overhead during the logging process and during recovery.

AVERAGE + STANDARD DEVIATION

LOGLIST provides the averages for the response time and block lengths. With the averages and the standard deviations which are also supplied, it is possible to determine your chances of attaining desired response times or block lengths. For instance, the interval covered by the sum of the average plus one standard deviation includes approximately 85% of all data base transactions logged.

DETAIL SET (DATA BASE) SUMMARY

The DETAIL SET summary provides totals based on the actual activity in the sets. As shown in Figure 20, this information includes the number of DBDELETES, DBPUTS, and DBUPDATES. The capacity and number of entries are also printed.

How LOGLIST Works

When processes are using the "USER LOGGING" facility of MPE, the process opens up a path to the transaction log for each process and each database enabled for logging. As part of this "opening" procedure the user's name, acct, process name, capability, LDEV, and database (if one) are logged in a special record. LOGLIST looks for these records and builds its internal working tables from them.

As processes make transactions to their databases, the logging process intercepts a copy of the changes, adds a time and date stamp then routes them to the logging file. LOGLIST uses the time stamp from the DBBEGIN and DBEND records to determine the total elapsed transaction time. (If you don't use DBBEGIN or DBENDS then you can never measure your response times with LOGLIST!)

Broken transactions can be located by looking for a special "ABNORMAL END" record, and by checking to make sure that all process issued a DBEND before closing the log and terminating.

If the process did not (or was unable) to close the log before terminating, and LOGLIST detects an EOF on the log then it is assumed that there has been a system crash. System crashes can also be determined by looking for the crash marker which was written out at the time of a WARMSTART recovery.

Transactions are expanded by using the information gathered when the process first opened up the log, and the actual data- base "change" records. (These records are marked with "DE," "PU" or "UP.") LOGLIST uses the item-list recorded as part of the transaction and calls DBINFO to determine the types and lengths of the individual items logged.

APPENDIX DBPERF

This program was written to benchmark the time required to perform a wide range of DBPUTS, DBDELETES and DBUPDATES. The primary area of interest was the overhead added to IMAGE/3000 when the user is using transaction logging.

The benchmark process follows the procedure listed below:

A. Disable the database XYZ for logging

B. Perform 50 DBOPEN's and DBCLOSE's to measure time to initially startup the logging process. (NOTE: this will really clobber the response time for everybody else.)

C. Perform 50 DBPUTS to a detail set which contains a single path and various data types. The data used for these operations is generated using the RAND function from the compiler library.

D. Perform 50 DBDELETES to the detail set.

E. Setup a loop so that we can perform 50 DBPUTS and DBDELETES on detail sets which contain from 0 to 15 paths.

F. Generate the plots and data summaries.

G. G) Enable database XYZ for logging, then repeat steps B) thru F)

The database modifications are performed without signaling the start of the transactions with DBBEGIN or the end with DBEND. This was done so that the comparison could be made, without the overhead added by the BEGIN-END blocking. This type of test is fair since the DBBEGIN and DBEND calls are made only to signify that that are changes which are dependent.

The time required to perform the BEGIN-END block is measured and plotted on a separate chart. It should be noted that since DBBEGIN and DBEND do not require immediate access to the disc drives, the time required to perform these intrinsics is very low. The can however add a significant number of records to the memory buffer, which of course means that there is an additionaly load on the I/O channel which controls the disc drives.

REFERENCES

[1]F. Alfredo Rego, "DATABASE THERAPY: A practitioner's experiences," in HPGSUG 1981 Orlando Florida Proceedings, Vol 1, pp. B12-01 to B12-13

[2]P. Sinclair, "MPE 1918: A BONANZA OF ENHANCEMENTS," in COMMUNICATOR issue 23, pp. 4-17

[3]HP, "MPE III 1918 USER UPDATE COURSE"

[4]HP, "MPE III Intrinsics Reference Manual," pp. 3-92 to 3-96

[5]HP, "IMAGE Data Base Management System reference manual," pp. 4-22 to 4-23

[6]HP, "IMAGE Data Base Management System reference manual," pp. 4-23

[7]P. Sinclair, "MPE 1918: A BONANZA OF ENHANCEMENTS," in COMMUNICATOR issue 23, pp. 14

[8]HP, "MPE III 1918 USER UPDATE COURSE," pp. 60

[9]Robert M. Green, Robelle Consulting Ltd., 5421 10th Avenue, Suite 130, Delta, British Columbia V4M 3T,. Canada.

[10]HP, "MPE III 1918 USER UPDATE COURSE," pp. 71

[11]A. O. Allen, "Queueing Models of Computer Systems," in COMPUTER, pp. 13-24, Apr. 1980 (an IEEE publication)

[12]C. Storla, "MEASURING TRANSACTION RESPONSE TIMES," in 1981 IUG Orlando Florida Proceedings, Vol. 1, pp. C7-01 to C7-08

```
========
= user =
=program
=      ==
======== =
          =
           =    ====================  =======  =====
========    =   =               = T  =  =     =  =   =
= user =     = =                = r  =  = 8   =  =   =
=program      = =    IMAGE      = a I =  = K B =  =   =
=   ==========                  = n o =  =   u =  = D =      ====
========      = =  data base    = s g =  = M f =  = i =     =      ===
             = =                = a g ===> e f ===> s ==>= Tape    =
            =  =  management    = c i =  = m e =  = s =   = ( if   =
========   =   =                = t n =  = o r =  =   =   =used)=
= user =  =    =    system      = i g =  = r   =  =   =    ====
=program =     =                = o   =  = y   =  =   =
=      ==      =                = n   =  =     =  =   =
========       ====================  =======  =====
                     =
                     =
                    \/
                ==========
                =  disc  =
                =  drive =
                ==========
```

Figure 1. IMAGE transaction logging flow

```
CALL DBBEGIN(BASE,...    )
     CALL DBLOCK(BASE,...  )

     CALL DBDELETE(BASE,... )

               o                       At this point,
               o                 if there is a crash we lose
               o                 this data entry!
               o

      change made to search item

               o
               o
               o

     CALL DBPUT(BASE,... )

               o              this item has now been re-added

     CALL DBUNLOCK(BASE,... )
CALL DBEND(BASE,...    )
```

Figure 2. Dependent Changes

```
CALL DBLOCK(... )
     CALL DBFIND(... )
     CALL DBGET(... )

     ** MAKE CHANGES TO ITEM VALUES HERE **

     CALL DBBEGIN(... )

                DBPUT
          CALL {DBUPDATE } (... )
                DBDELETE

     CALL DBEND(... )
CALL DBUNLOCK(... )
```

Figure 3A

```
CALL DBFIND(... )
CALL DBGET(... )

** MAKE CHANGES TO ITEM VALUES HERE **

     CALL DBBEGIN(... )
     CALL DBLOCK(... )

               DBPUT
          CALL{DBUPDATE}(... )
               DBDELETE

     CALL DBUNLOCK(... )
     CALL DBEND(... )
```

Figure 3B

```
CALL BEGINLOG(... )

     CALL DBFIND(BASE1,... )
     CALL DBGET(BASE1,... )

     ** MAKE CHANGES TO ITEM VALUES HERE **

          CALL DBBEGIN(BASE1,... )
          CALL DBLOCK(BASE1,... )

                    DBPUT
               CALL{DBUPDATE}(BASE1,... )
                    DBDELETE

          CALL DBUNLOCK(BASE1,... )
          CALL DBEND(BASE1,... )

     CALL DBFIND(BASE2,... )
     CALL DBGET(BASE2,... )

     ** MAKE CHANGES TO ITEM VALUES HERE **

          CALL DBBEGIN(BASE2,... )
          CALL DBLOCK(BASE2,... )

                    DBPUT
               CALL{DBUPDATE}(BASE2,... )
                    DBDELETE

          CALL DBUNLOCK(BASE2,... )
          CALL DBEND(BASE2,... )

CALL ENDLOG(... )
```

Figure 4

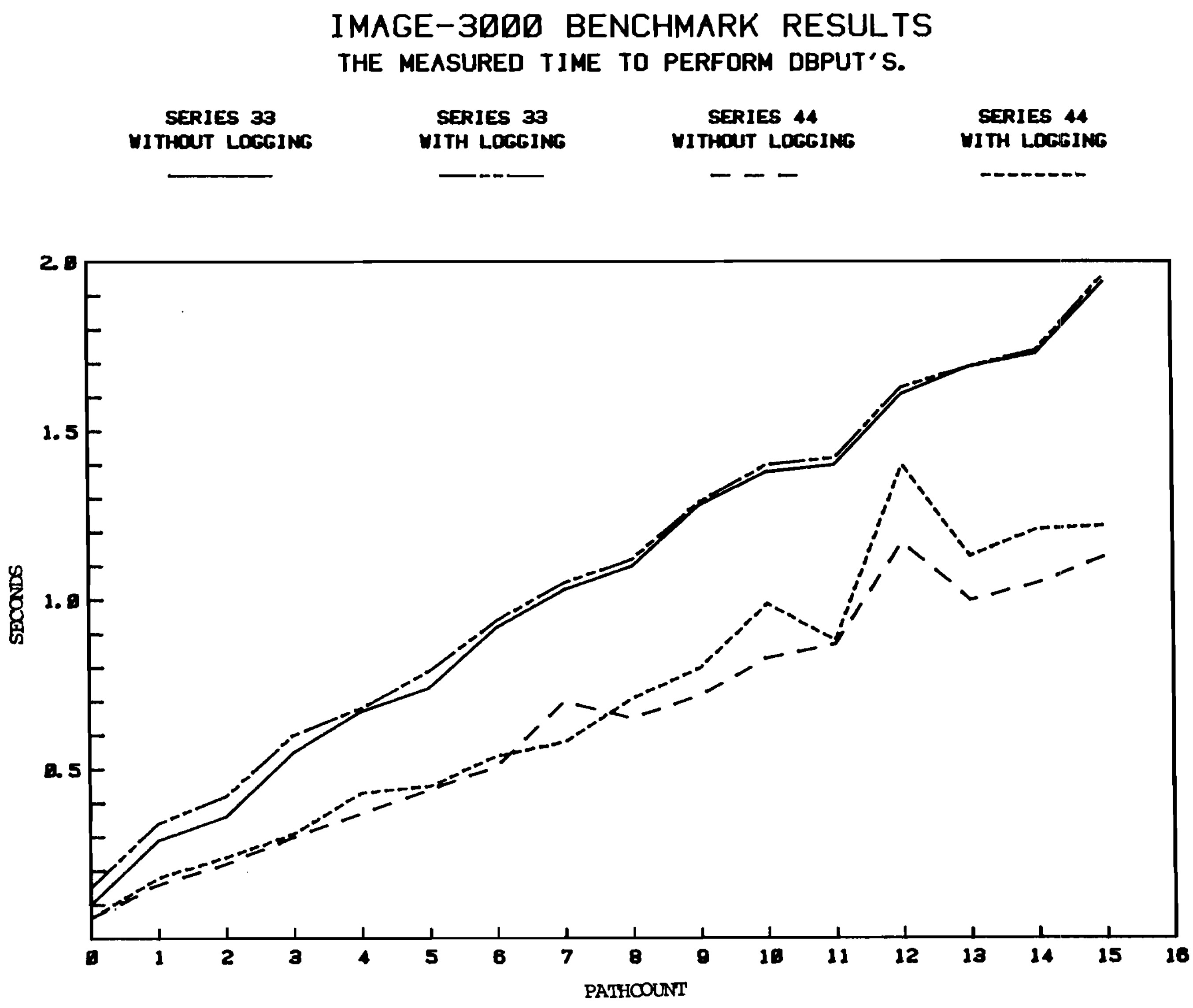


IMAGE-3ØØØ BENCHMARK RESULTS
THE MEASURED TIME TO PERFORM DBPUT'S.
SERIES 33 WITHOUT LOGGING
SERIES 33 WITH LOGGING
SERIES 44 WITHOUT LOGGING
SERIES 44 WITH LOGGING
2.Ø
1.5
1.Ø
Ø.5
SECONDS
Ø 1 2 3 4 5 6 7 8 9 1Ø 11 12 13 14 15 16
PATHCOUNT

Figure 5

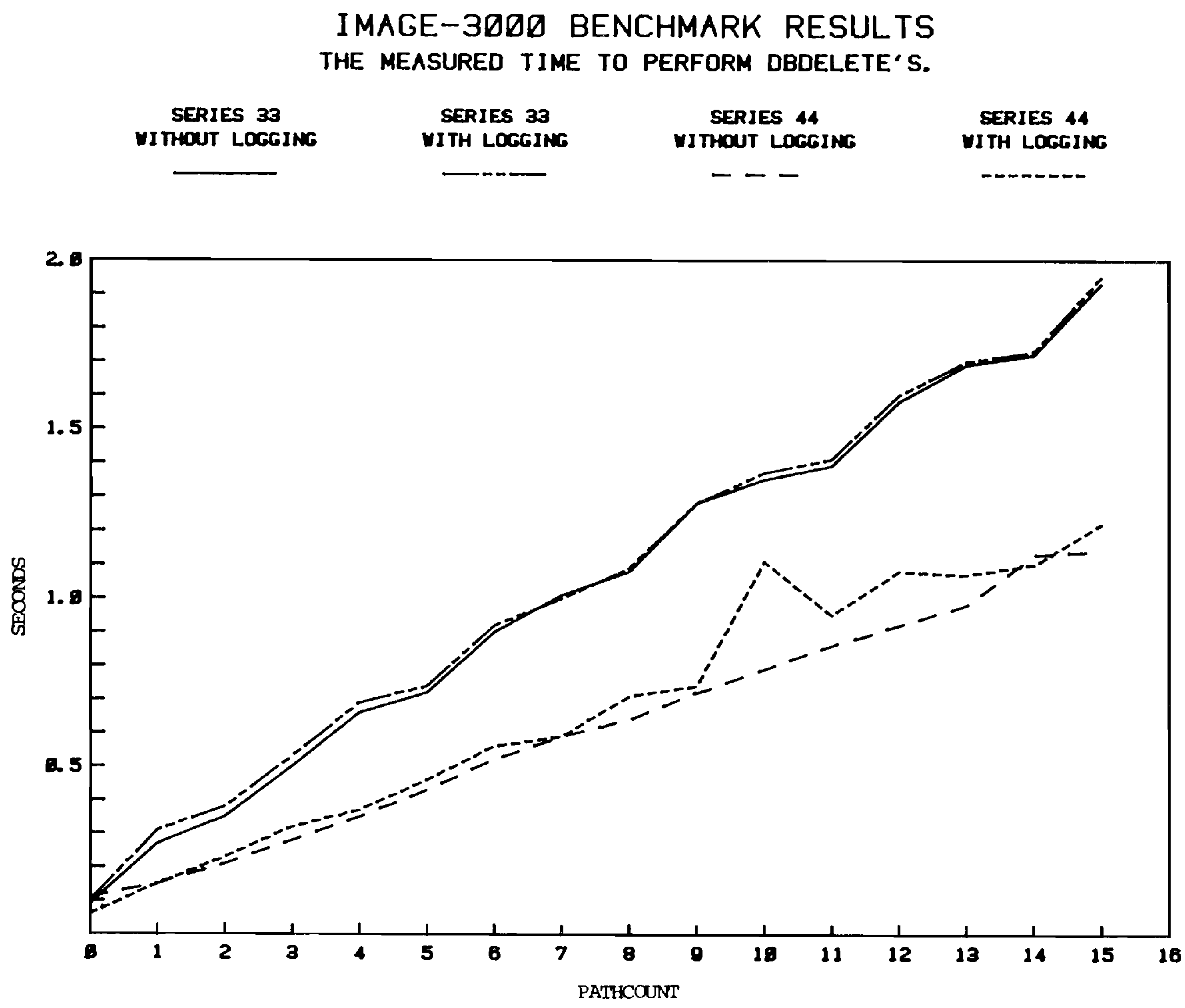


IMAGE-3000 BENCHMARK RESULTS
THE MEASURED TIME TO PERFORM DBDELETE'S.
SERIES 33 WITHOUT LOGGING
SERIES 33 WITH LOGGING
SERIES 44 WITHOUT LOGGING
SERIES 44 WITH LOGGING
2.0
1.5
1.0
0.5
SECONDS
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
PATHCOUNT

Figure 6

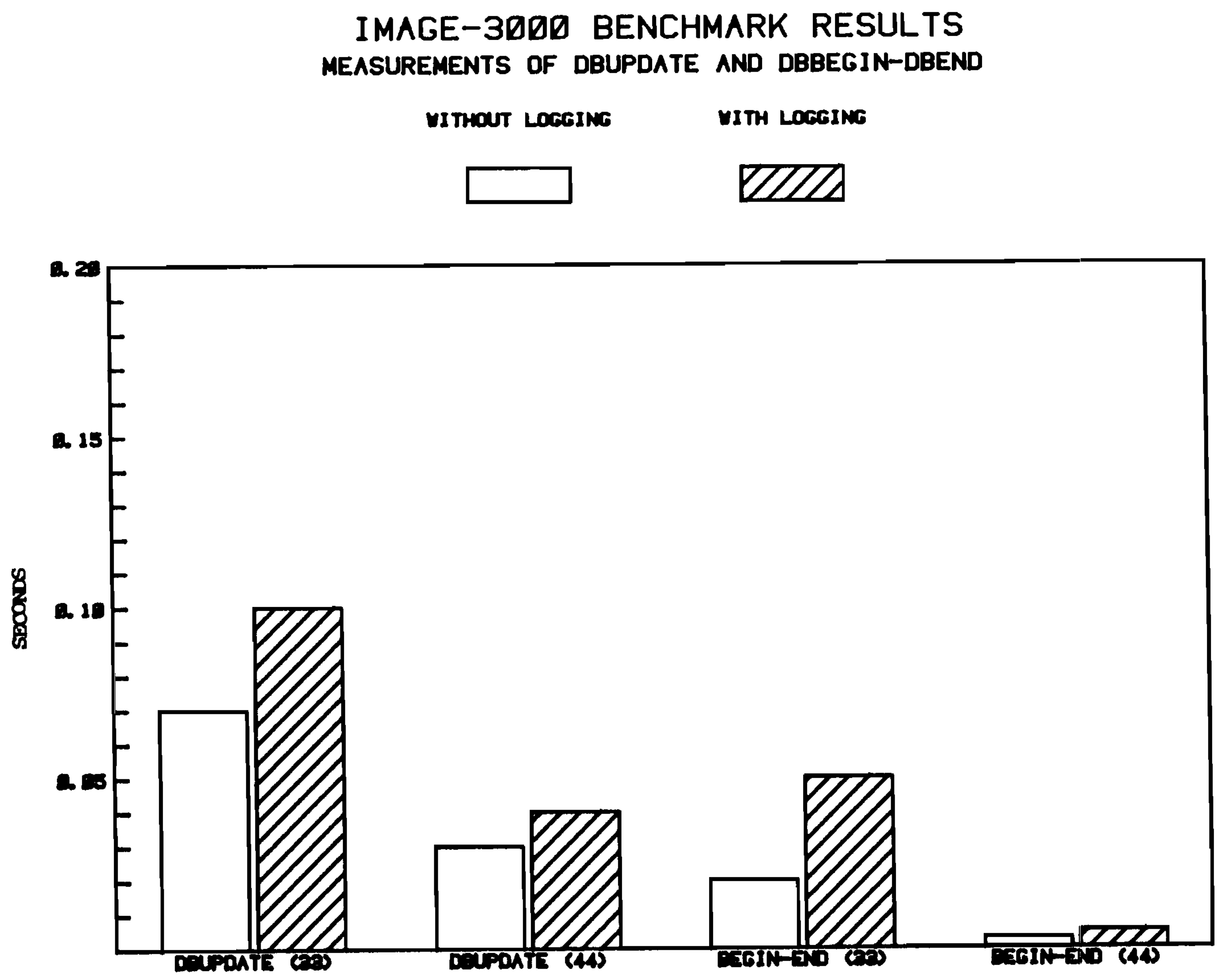


IMAGE-3000 BENCHMARK RESULTS
MEASUREMENTS OF DBUPDATE AND DBBEGIN-DBEND
WITHOUT LOGGING
WITH LOGGING
0.20
0.15
0.10
0.05
SECONDS
DBUPDATE (22)
DBUPDATE (44)
BEGIN-END (22)
BEGIN-END (44)

Figure 7

```
CALL DBBEGIN(BASE,...    )
     CALL DBLOCK(BASE,...  )

     CALL DBDELETE(BASE,... ) <structural damage,if crash occurrs>
                              < for a detail set with 5 paths >
                              < the 'critical' time could be  >
                              < a half second or more!        >

               o                    At this point,
               o                 if there is a crash we lose
               o                 this data entry!
               o

      change made to search item

               o
               o
               o

     CALL DBPUT(BASE,... ) <structural damage, if crash occurrs>
                              < for a detail set with 5 paths >
                              < the 'critical' time could be  >
                              < a half second or more!        >

               o                 The item has now be re-added

     CALL DBUNLOCK(BASE,... )
CALL DBEND(BASE,...    )
```

Figure 8. Crash Modes

```
         Benefits of Logging    VS    Cost of Logging


AUDIT TRAIL                             REDUCED THROUGHPUT?
 Who, What, When and How                 Dependent on your
 Ability to list Sets and                application, and system
 fields which are modified.              load.

RECOVERABLE DATA                        COST OF ADDITIONAL MEMORY?
 Ability to recover                      May need more memory
 most if not all                         to maintain current
 transactions, upto the                    system throughput.
 time of the crash.

PERFORMANCE INFO                        TAPE DRIVE OR DISC
 Information available                  DEDICATED TO LOGGING?
 which can lead to better                Valuable disc space
 application designs                     or tape drive can be
 in the future.                          tied up with logging.

LOGGING OF ALL CHANGES                  STARTUP AFTER CRASH
MADE TO DATA BASE                       MORE COMPLICATED
REGARDLESS OF PROGRAM
 You can use any vendor                  Training and "test
 software and still                      recoverys" may be
 maintain an "audit trail".              required to familiarize
                                         the programmers and
                                         operators with the
                                         new system restart
HP  SUPPORT OF LOGGING                   procedures.
============================================================
                            ^
                           / \
                          /   \
                         /     \
                        =========
```

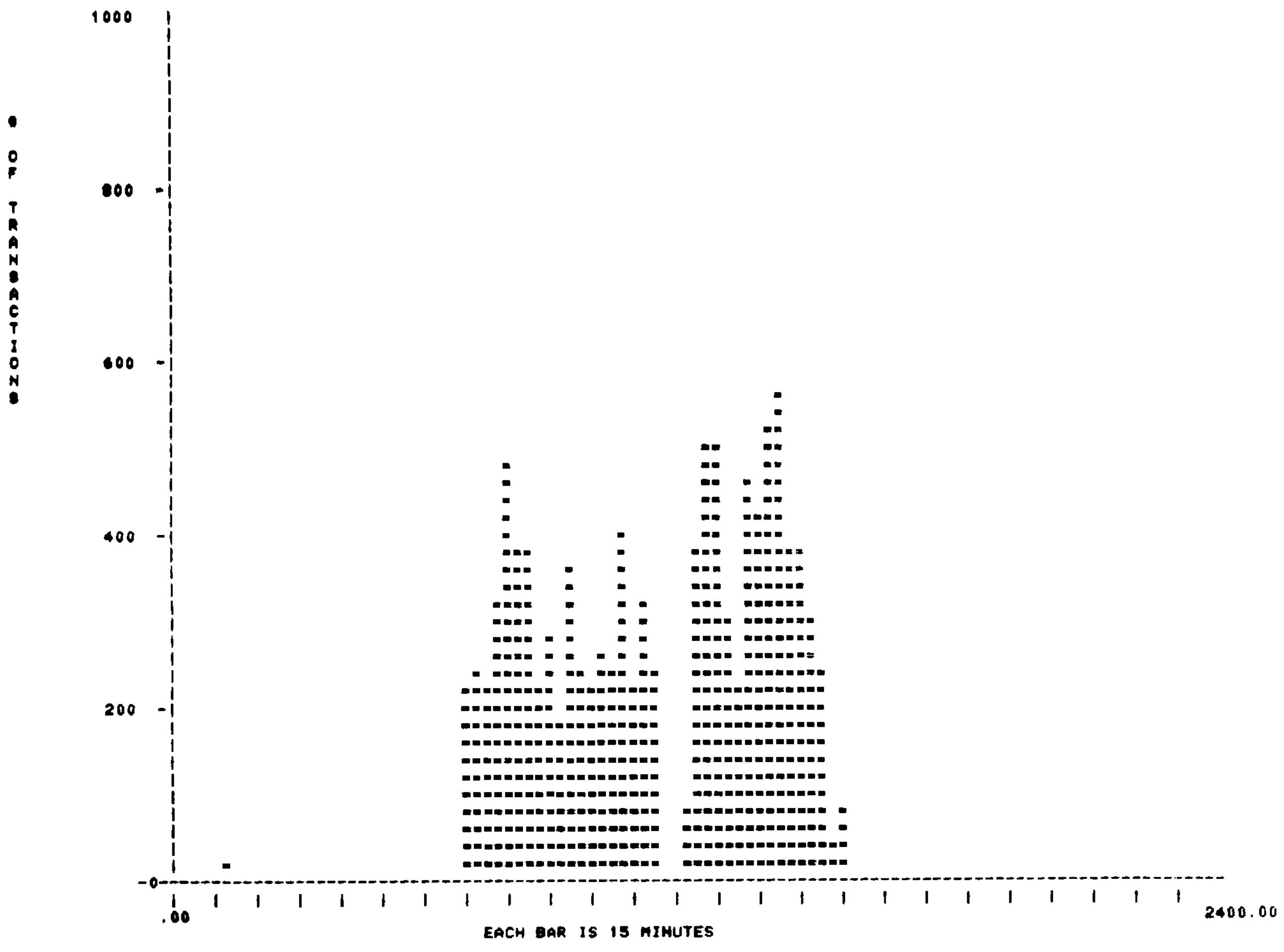
Figure 9

```
****************************************************************************************************
*     OFFICE  BAC      TEIMS          ACCTPROGBAC      TEIMS          TUE, DEC 29, 1981,  8:58 PM        *
*  1                   [illegible]                                                              *
*                                                                                                   *
*          LOGID#:   1  TRANSACTION# -      4   DELETEING ITEM IN      EQUIP-DETL         DBFILE RECORD#: 23277    *
*      PROP#           [U12  ] = 1                                                                  *
*      MODELCOD        [U14  ] = 0003FIDDLE                                                         *
*      NOMNCODE        [I1   ] = 3                                                                  *
*      EQUIPLOC        [I1   ] = 10                                                                 *
*      PROGTAG         [I1   ] = 0                                                                  *
*      CURRUSER        [U10  ] = 0000000100                                                         *
*      NEXTUSE1        [U10  ] = NONE                                                               *
*      PO-SERIES       [U10  ] = NONE                                                               *
*      HOLDFOR         [U10  ] = NONE                                                               *
*      BORROWER        [U10  ] = NONE                                                               *
*      FROMUSER        [U10  ] = NONE                                                               *
*      MFG             [I1   ] = 0                                                                  *
*      INVTORYTAG      [I1   ] = 0                                                                  *
*      CALDATE         [I2   ] = 811225                                                             *
*      SERIAL#         [U10  ] = SERIAL                                                             *
*      ACQUCOST        [I2   ] = 11                                                                 *
*      CAPEXPEN        [U2   ] = E                                                                  *
*      NEWUSED         [U2   ] = N                                                                  *
*      ISSUEDAT        [I2   ] = 811220                                                             *
*      ADCODE          [U2   ] = AC                                                                 *
*      ACQUDATE        [I2   ] = 811115                                                             *
*      NEXTSTD1        [I2   ] = 0                                                                  *
*      NEXTSTD2        [I2   ] = 0                                                                  *
*      HOLDATE         [I2   ] = 0                                                                  *
*      NEXTEND1        [I2   ] = 0                                                                  *
*      NEXTEND2        [I2   ] = 0                                                                  *
*      STARTCYCLE      [I2   ] = 0                                                                  *
*      RETNDATE        [I2   ] = 0                                                                  *
*      BORROWDT        [I2   ] = 0                                                                  *
*      MESSTAG1        [I1   ] = 0                                                                  *
*      MESSTAG2        [I1   ] = 0                                                                  *
*      MESSTAG3        [I1   ] = 0                                                                  *
*      ACCESTAG        [I1   ] = 0                                                                  *
*      OPTINTAG        [X2   ] = AA                                                                 *
*      SPECCODE        [I1   ] = 0                                                                  *
*      SAMPUTIL        [U2   ] = YE                                                                 *
*      LASTUSER        [U10  ] = 5176822150                                                         *
*          LOGID#:   1  TRANSACTION# -      4   ADDING ITEM TO         EQUIP-DETL         DBFILE RECORD#: 23277    *
*      PROP#           [U12  ] = 1                                                                  *
*      MODELCOD        [U14  ] = 0003FIDDLE                                                         *
*      NOMNCODE        [I1   ] = 3                                                                  *
*      EQUIPLOC        [I1   ] = 10                                                                 *
*      PROGTAG         [I1   ] = 0                                                                  *
*      CURRUSER        [U10  ] = 5172870970                                                         *
*      NEXTUSE1        [U10  ] = NONE                                                               *
*      PO-SERIES       [U10  ] = NONE                                                               *
*      HOLDFOR         [U10  ] = NONE                                                               *
*      BORROWER        [U10  ] = NONE                                                               *
*      FROMUSER        [U10  ] = NONE                                                               *
*      MFG             [I1   ] = 0                                                                  *
*      INVTORYTAG      [I1   ] = 30003                                                              *
*      CALDATE         [I2   ] = 811225                                                             *
*      SERIAL#         [U10  ] = SERIAL                                                             *
*      ACQUCOST        [I2   ] = 11                                                                 *
*      CAPEXPEN        [U2   ] = E                                                                  *
*      NEWUSED         [U2   ] = N                                                                  *
*      ISSUEDAT        [I2   ] = 811220                                                             *
*      ADCODE          [U2   ] = AC                                                                 *
*      ACQUDATE        [I2   ] = 811115                                                             *
*      NEXTSTD1        [I2   ] = 0                                                                  *
*      NEXTSTD2        [I2   ] = 0                                                                  *
*      HOLDATE         [I2   ] = 0                                                                  *
*      NEXTEND1        [I2   ] = 0                                                                  *
*      NEXTEND2        [I2   ] = 0                                                                  *
*      STARTCYCLE      [I2   ] = 0                                                                  *
*      RETNDATE        [I2   ] = 0                                                                  *
*      BORROWDT        [I2   ] = 0                                                                  *
*      MESSTAG1        [I1   ] = 0                                                                  *
*      MESSTAG2        [I1   ] = 0                                                                  *
*      MESSTAG3        [I1   ] = 0                                                                  *
*      ACCESTAG        [I1   ] = 0                                                                  *
*      OPTINTAG        [X2   ] = AA                                                                 *
*      SPECCODE        [I1   ] = 0                                                                  *
*      SAMPUTIL        [U2   ] = YE                                                                 *
*      LASTUSER        [U10  ] = 5176822150                                                         *
*                      8:58 PM                                                                *     *
****************************************************************************************************

****************************************************************************************************
*     OFFICE  BAC      TEIMS          ACCTPROGBAC      TEIMS          TUE, DEC 29, 1981,  8:59 PM        *
*                                                                                                   *
*                                                                                                   *
*          LOGID#:   1  TRANSACTION# -      5   UPDATING ITEM IN       EQUIP-DETL         DBFILE RECORD#: 23277    *
*      NEW VALUES:                                                                                  *
*      CALDATE         [I2   ] = 0                                                                  *
*      ADCODE          [U2   ] = NA                                                                 *
*                                                                                                   *
*      OLD VALUES:                                                                                  *
*      CALDATE         [I2   ] = 811225                                                             *
*      ADCODE          [U2   ] = AC                                                                 *
*                      8:59 PM                                                                *     *
****************************************************************************************************

          LOGID#:   1  TRANSACTION# -      1   UPDATING ITEM IN       EQUIP-DETL         DBFILE RECORD#: 23277
      NEW VALUES:
      MFG             [I1   ] = 3

      OLD VALUES:
      MFG             [I1   ] = 0
```

Figure 10

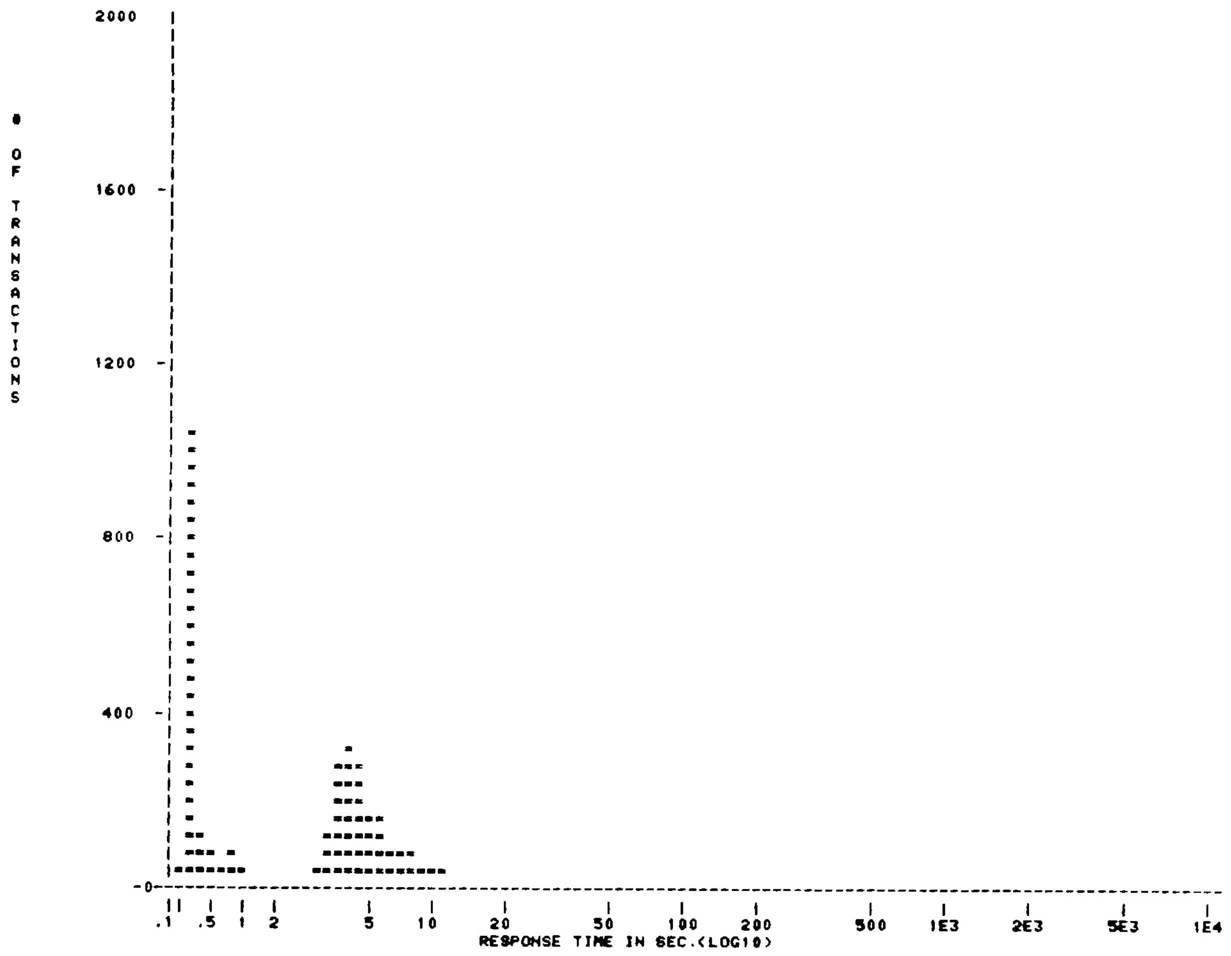
```
  USER    GROUP   ACCT  DBASE PROCESS  GROUP   ACCT          LOGON TIME           LOGOFF TIME  LG# DEV O CAPABILITY   UP  PUT  DEL #BLKS
--------------------------------------------------------------------------------------------------------------------------------------

OFFICE  BAC      TEIMS   TEIM1 QUERY   PUB     SYS     WED,NOV 25,1981, 1:55P NOV 25, 2:02P 163 25 1 4020300611    1    0    0     0
BML     BAC      TEIMS   TEIM1 HAPROG  BAC     TEIMS   WED,NOV 25,1981, 1:50P NOV 25, 2:04P 162 37 1 0060300601    9    3    3     3

         WED, NOV 25, 1981,  2:13 PM  *** PROCESS ABORTED ***  PLTII    BAC      TEIMS
PLTII   BAC      TEIMS   TEIM1 HAPROG  BAC     TEIMS   WED,NOV 25,1981, 2:05P NOV 25, 2:13P 164 36 1 0020300611    3    1    1    1?
OFFICE  BAC      TEIMS   TEIM1 QUERY   PUB     SYS     WED,NOV 25,1981, 2:07P NOV 25, 2:14P 165 25 1 4020300611    1    0    0     0
DC      BAC      TEIMS   TEIM1 HAPROG  BAC     TEIMS   WED,NOV 25,1981, 8:01A NOV 25, 2:17P 132 34 1 0020300611   98   50   38    38
BAC     BAC      TEIMS   TEIM1 ACCTST  BAC     TEIMS   WED,NOV 25,1981, 2:19P NOV 25, 2:20P 169 26 1 5360300613    0    0    0     0
DC      BAC      TEIMS   TEIM1 HAPROG  BAC     TEIMS   WED,NOV 25,1981, 6:55A NOV 25, 2:36P 127 39 1 0020300611  114   54   42    42
PRIMARY BAC      TEIMS   TEIM1 QAPROG  BAC     TEIMS   WED,NOV 25,1981, 2:09P NOV 25, 2:38P 166 37 1 0020300611   18    6    6     6
PRIMARY BAC      TEIMS   TOOLS QAPROG  BAC     TEIMS   WED,NOV 25,1981, 2:09P NOV 25, 2:38P 167 37 1 0020300611    0    0    0    0*
QA      BAC      TEIMS   TEIM1 QAPROG  BAC     TEIMS   WED,NOV 25,1981, 6:32A NOV 25, 2:52P 122 32 1 0020300611  251   99   99    99
QA      BAC      TEIMS   TOOLS QAPROG  BAC     TEIMS   WED,NOV 25,1981, 6:32A NOV 25, 2:52P 123 32 1 0020300611    0    0    0    0*
QA      BAC      TEIMS   TEIM1 QAPROG  BAC     TEIMS   WED,NOV 25,1981, 2:53P NOV 25, 2:56P 172 32 1 0020300611    0    0    0     0
QA      BAC      TEIMS   TOOLS QAPROG  BAC     TEIMS   WED,NOV 25,1981, 2:53P NOV 25, 2:56P 173 32 1 0020300611    0    0    0    0*
QA      BAC      TEIMS   TEIM1 QAPROG  BAC     TEIMS   WED,NOV 25,1981, 2:58P NOV 25, 2:59P 174 32 1 0020300611    0    0    0     0
QA      BAC      TEIMS   TOOLS QAPROG  BAC     TEIMS   WED,NOV 25,1981, 2:58P NOV 25, 2:59P 175 32 1 0020300611    0    0    0    0*
BAC     BAC      TEIMS   TEIM1 QUERY   PUB     SYS     WED,NOV 25,1981, 2:43P NOV 25, 3:04P 171 26 1 5360300613    0    2    1     0
PLTII   BAC      TEIMS   TEIM1 HAPROG  BAC     TEIMS   WED,NOV 25,1981, 2:16P NOV 25, 3:05P 168 36 1 0020300611   49   23   18    18
KENT    BAC      TEIMS   TEIM1 HAPROG  BAC     TEIMS   WED,NOV 25,1981, 6:39A NOV 25, 3:08P 124 33 1 0020300611  220   97   78    83
DC      BAC      TEIMS   TEIM1 HAPROG  BAC     TEIMS   WED,NOV 25,1981, 2:37P NOV 25, 3:20P 170 34 1 0020300611   16    8    6     6
KENT    BAC      TEIMS   TEIM1 NAPROG  BAC     TEIMS   WED,NOV 25,1981, 6:26A NOV 25, 3:24P 120 31 1 0020300611  185   99   71    71
KENT    BAC      TEIMS   TEIM1 HAPROG  BAC     TEIMS   WED,NOV 25,1981, 3:09P NOV 25, 3:26P 177 32 1 0020300611   12    8    5     5
BAC     BAC      TEIMS   TEIM1 ACCTPROGBAC     TEIMS   WED,NOV 25,1981, 3:04P NOV 25, 3:39P 176 26 1 5360300613    4    1    1     4

         WED, NOV 25, 1981,  3:44 PM  *** PROCESS ABORTED ***  TEKSTAFFBAC      TEIMS
TEKSTAFFBAC      TEIMS   TEIM1 TECHPROGBAC     TEIMS   WED,NOV 25,1981,12:58P NOV 25, 3:44P 159 21 1 0020300611  109   20    1   98?
OFFICE  BAC      TEIMS   TEIM1 ACCTPROGBAC     TEIMS   WED,NOV 25,1981,12:13P NOV 25, 3:47P 152 25 1 4020300611  136   76   73   212
BAC     BAC      TEIMS   TEIM1 ACCTST  BAC     TEIMS   WED,NOV 25,1981, 3:55P NOV 25, 4:00P 178 26 1 5360300613    0    0    0     0
BAC     BAC      TEIMS   TEIM1 HAPROG  BAC     TEIMS   WED,NOV 25,1981, 4:27P NOV 25, 4:45P 179 31 1 5360300613    0    0    0     0
                     * - INDICATES PROCESS DID NOT QUALIFY IN SELECTIVE SEARCH     ? - BROKEN TRANSACTION
```

Figure 11



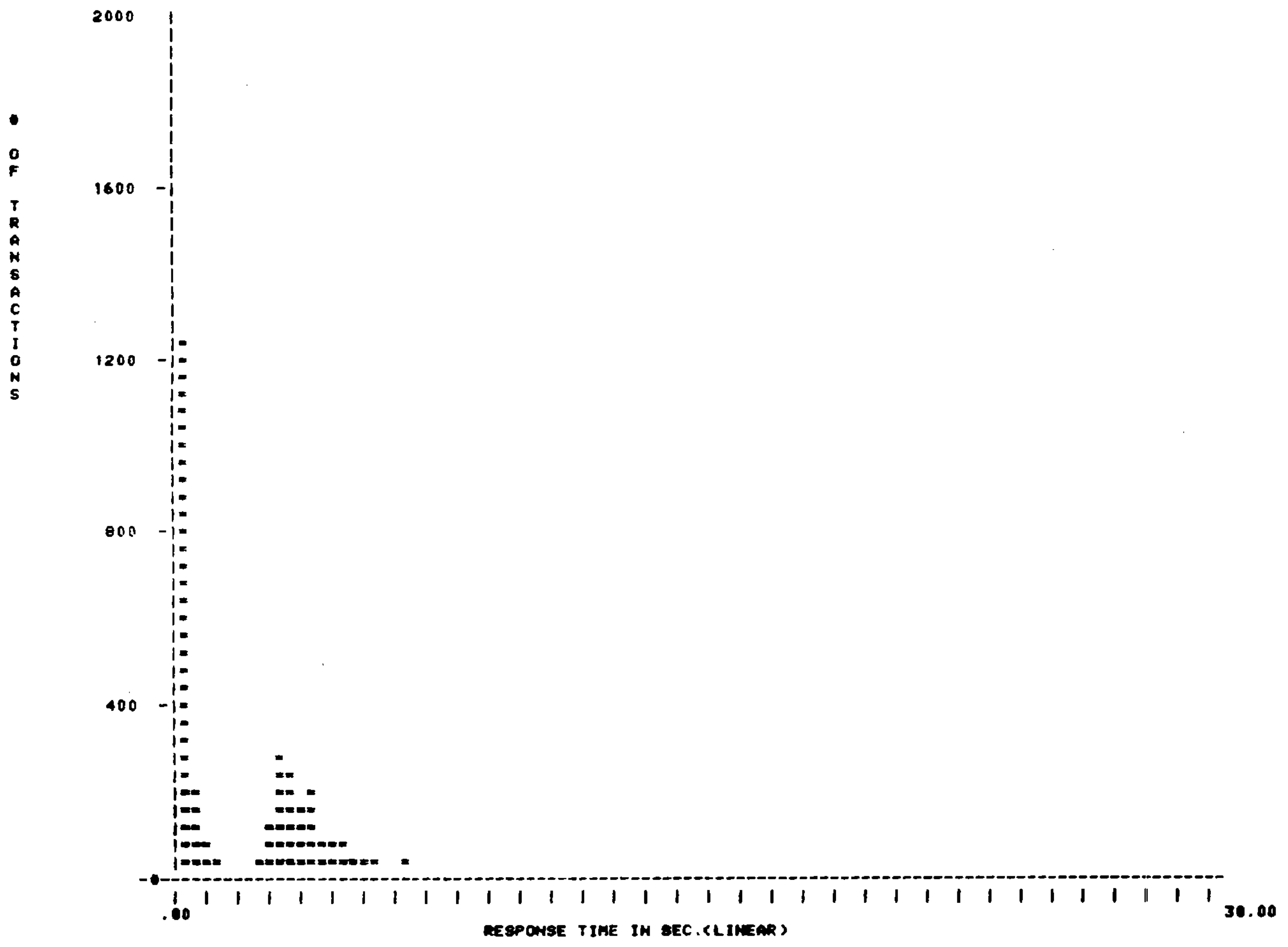
```
MAXIMUM VALUE:   560.0 MIN IS :     .0 SCALE FACTOR:   20.0 AVG :   111.86 Y AXIS MAX: 1000 TOTAL # IN ALL CELLS: 11186
```

Figure 12



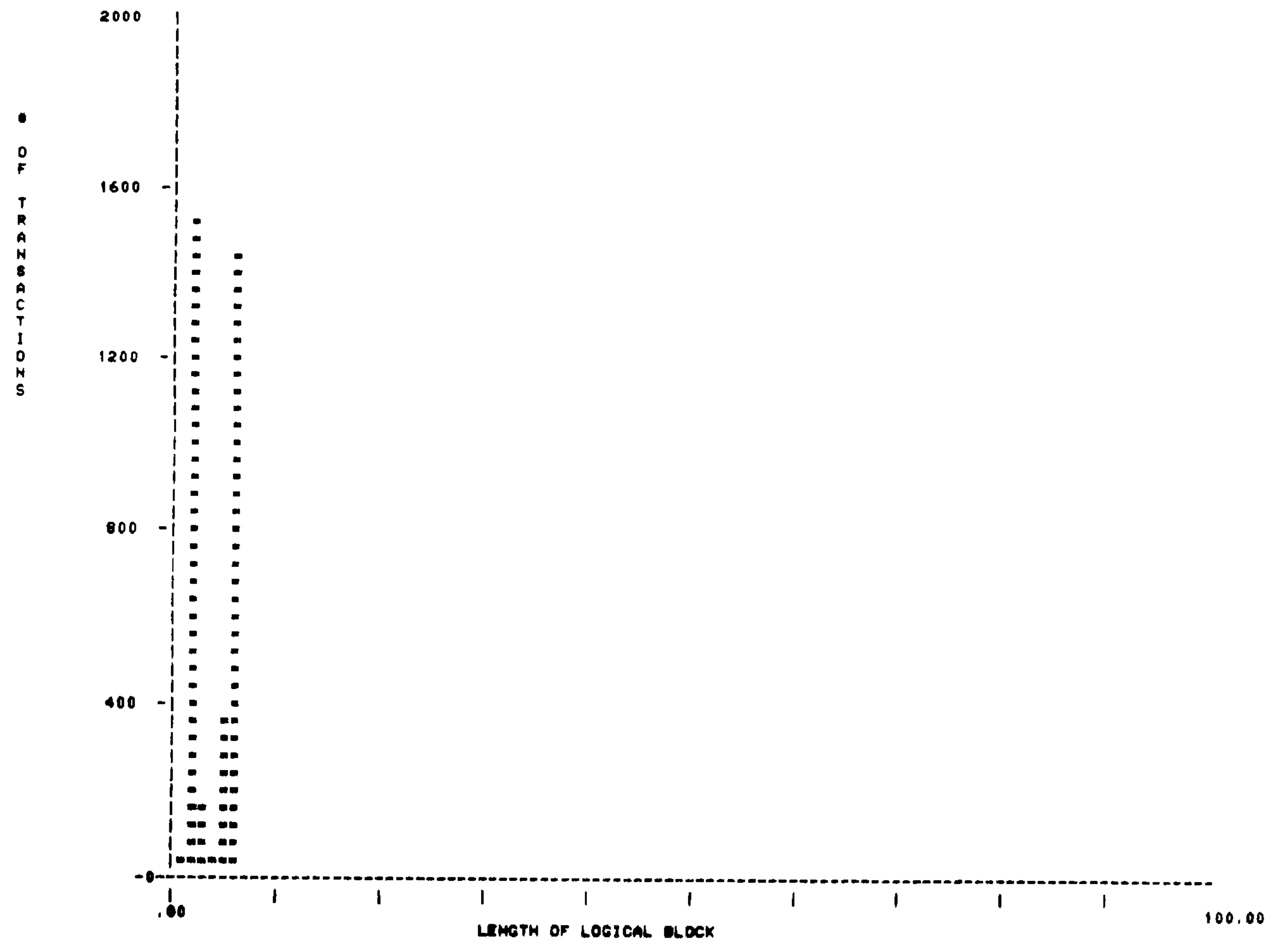
```
MAXIMUM VALUE:  1066.0 MIN IS :      .0 SCALE FACTOR:   40.0 AVG :    37.06 Y AXIS MAX: 2000 TOTAL # IN ALL CELLS:  3706
```

Figure 13



MAXIMUM VALUE: 1272.0 MIN IS : .0 SCALE FACTOR: 40.0 AVG : 37.06 Y AXIS MAX: 2000 TOTAL # IN ALL CELLS: 3706

Figure 14



```
MAXIMUM VALUE:  1538.0 MIN IS :      0 SCALE FACTOR:   40.0 AVG :   37.06 Y AXIS MAX: 2000 TOTAL # IN ALL CELLS:  3706
```

Figure 15

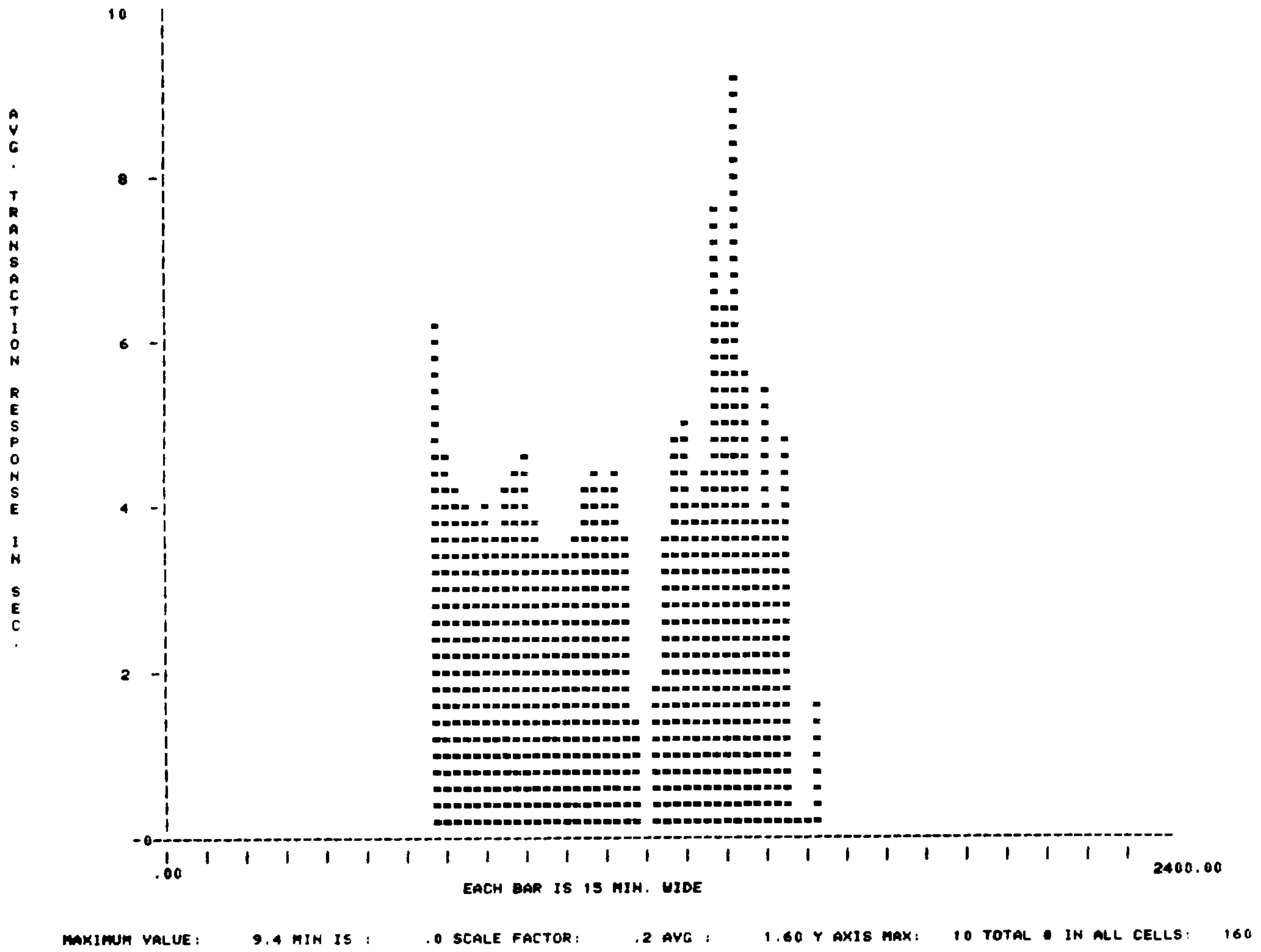
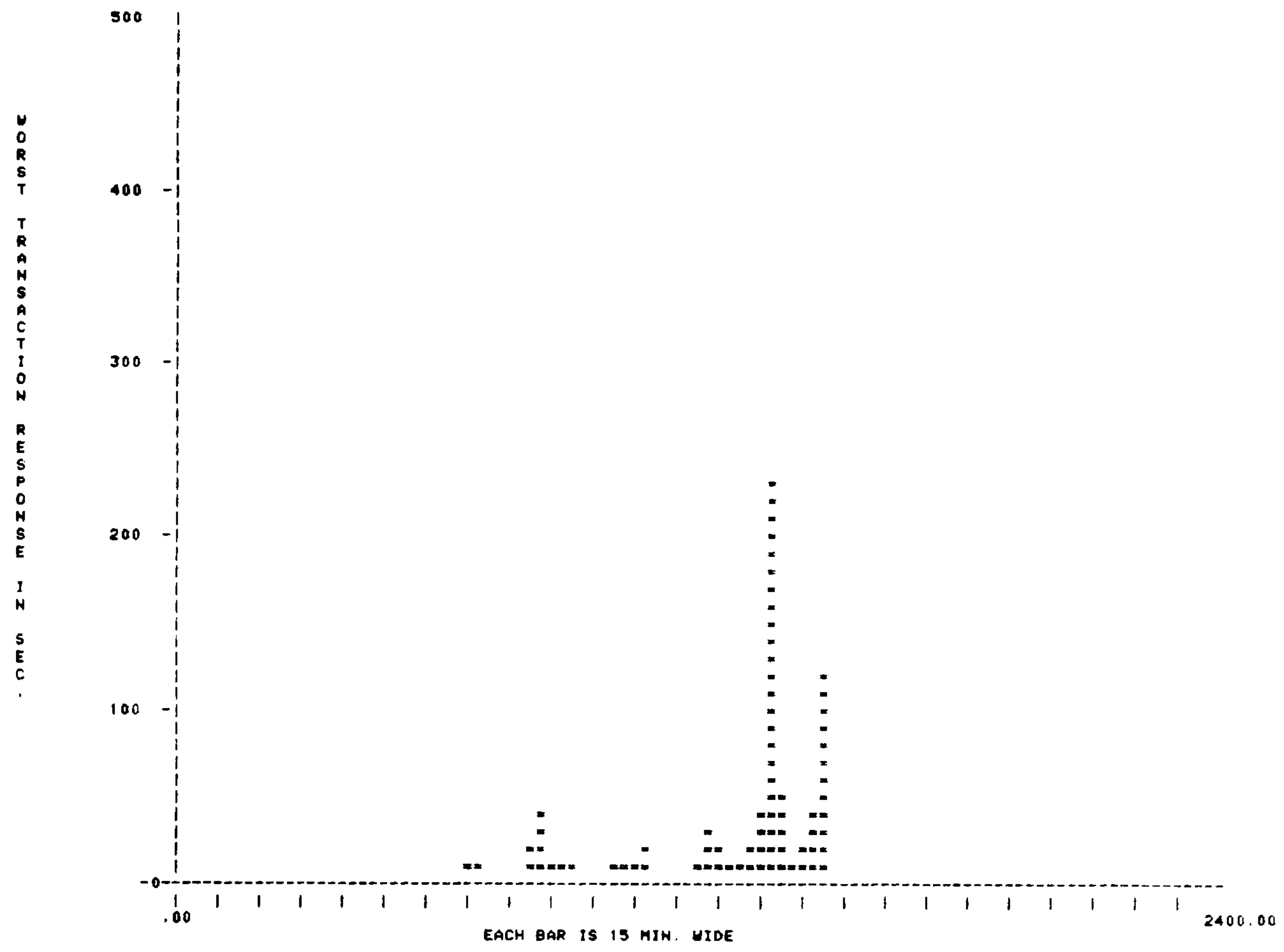
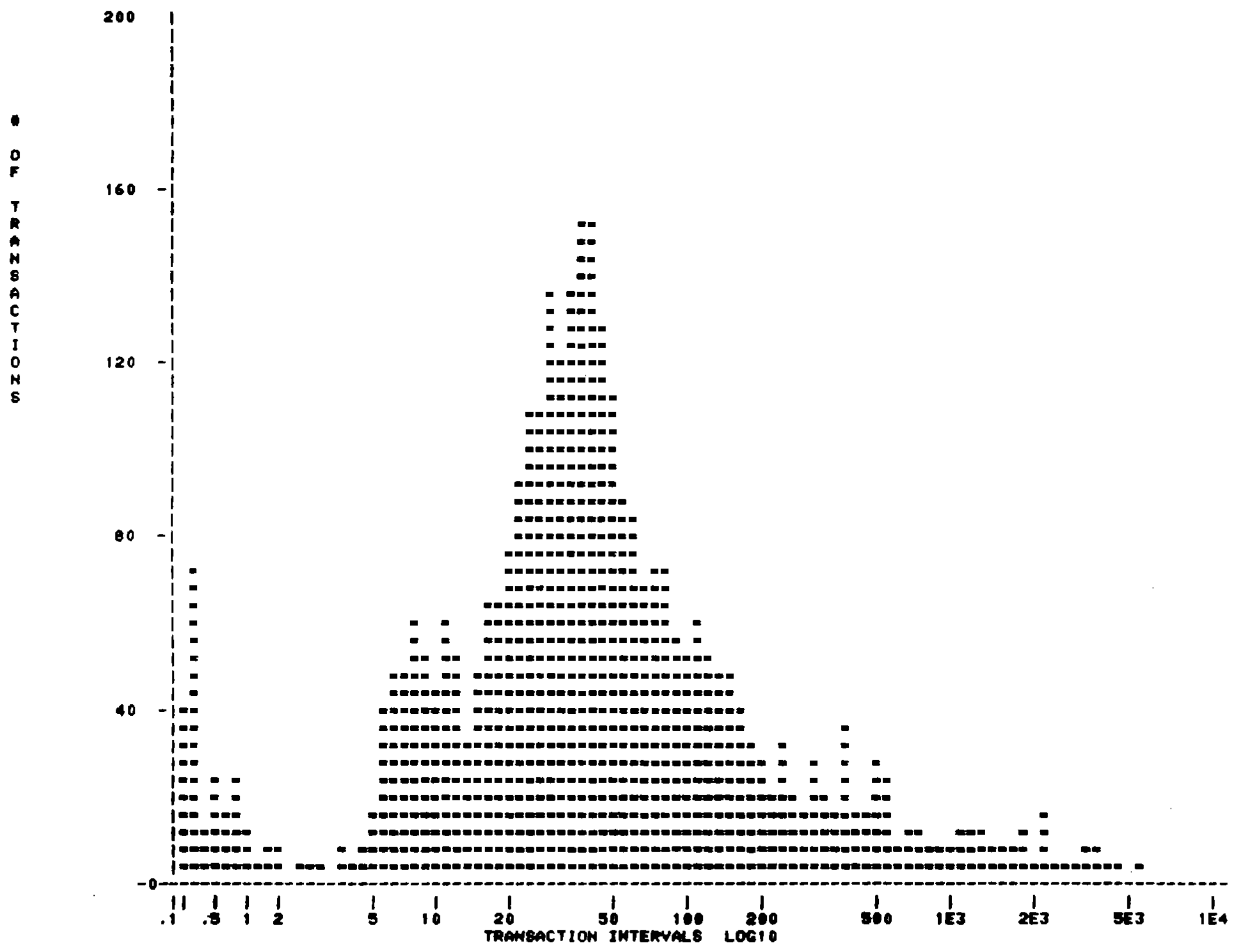

AVG. TRANSACTION RESPONSE IN SEC.
10
8
6
4
2
0
.00
2400.00
EACH BAR IS 15 MIN. WIDE
MAXIMUM VALUE: 9.4 MIN IS : .0 SCALE FACTOR: .2 AVG : 1.60 Y AXIS MAX: 10 TOTAL # IN ALL CELLS: 160

Figure 16



```
MAXIMUM VALUE:   235.4 MIN IS :      .0 SCALE FACTOR.   10.0 AVG :      9.87 Y AXIS MAX:  500 TOTAL # IN ALL CELLS.   987
```

Figure 17



```
MAXIMUM VALUE:   155.0 MIN IS :      .0 SCALE FACTOR:    4.0 AVG :    36.25 Y AXIS MAX:  200 TOTAL # IN ALL CELLS:  3625
```

Figure 18

```
***********************************************************
*      NUMBER OF RECORDS PROCESS        22112             *
*      PUTS, DELETES, UPDATES           11186             *
*      DBBEGINS & DBENDS                 7415             *
*      AVERAGE TRANSACTION TIME          1.60             *
*                STD DEVIATION           4.57             *
*                                                         *
*      AVERAGE TRANSACTION INTERVAL    182.04             *
*             STD DEVIATION            645.68             *
*                                                         *
*      AVERAGE BLOCK LENGTH              1.49             *
*             STD DEVIATION              2.00             *
*      # OF LOGICAL BLOCKS               3625             *
***********************************************************
```

Figure 19

```
DATA-SET(BASE)       #UPDATES  #DELETES  #PUTS    CAPACITY  ENTRIES  PERCENT FULL
---------------------------------------------------------------------------------

CROSSREFJOS                 0         8      7       50036    41355         82.65
OPTION-DETL                 0         0      1         987      369         37.39
SERV-DETL                 210       120    129       30984    21705         70.05
UTIL-DETL                 759        45    532       30990     7443         24.02
EQUIP-DETL                704      2015   2021       34986    27363         78.21
NOMCL-DETL                399         1     28        5004     3970         79.34
USER-DETL                3732         0      0        1704     1140         66.90
NOMH-DETL                  12         1     13        1512      697         46.10
SPEC-DETL                 229         5     48       34986    16389         46.84
USEWITH-XREF                0        10    106        1014      101          9.96
WHAREHOUSELOC               0        24     27        5031     2518         50.05
```

Figure 20

Modular Programming in MPE

Ingenieurbüro Jörg Grössler
IJG, Gbgh, Berlin

MODULAR PROGRAMMING

— There is no final definition yet

— A module can be embedded into any environment knowing its interface but not the algorithm used.

example:

sin (×)

the user must know:
- — x must be of type "REAL"
- — sin (×) will be of type "REAL"
- — sin (3.1415) = 0
- — $1.2E-50 < \times < 4.5E+55$
- — what happens in case of error

the user must not know:
- — the method how sin (×) is calculated

SOME MORE ASPECTS

— A module can be constructed without knowing the environment it will be used in

— The module interfaces should be as simple as possible

WHAT MODULES CAN OFFER

— Procedures
e.g.: sin(×)

— Data
e.g.: INTEGER ARRAY A

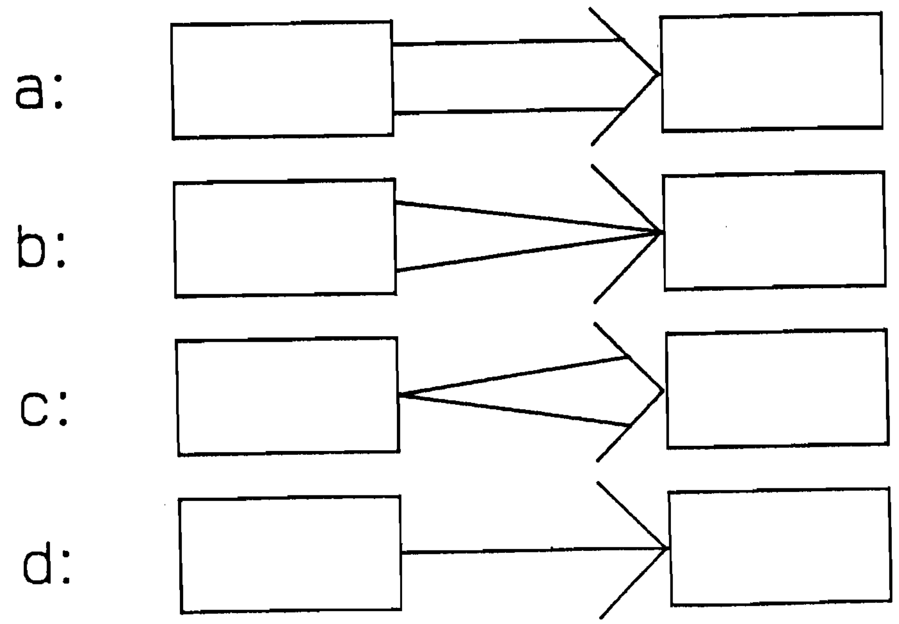
— Files
e.g.: Data-base

— Any mixture of the three above

MODULE INTERFACES

— Information flow between modules

— Described by:
- — The type of information (data, procedure, file)
- — The access rights for each communication direction:



Examples for Module Interfaces

```
a:   BEGIN
        INTEGER I;
        ...
        PROCEDURE P1;
          BEGIN
            I:=0;
            WHILE (I:=I+1) < 10 DO
               BEGIN .... END;
          END;
        ...
        I: *EQL 0;
        WHILE (I: =I+1) < 10 DO
          BEGIN
            ...
            P1;
          END;
        ...
     END;
c:          SUBROUTINE SUB
            ...
            INVAL=ITEMP (10)
            ...
            END
```

MODULE REQUIREMENTS

— Control of information flow (specification of imported and exported objects)

— Check of interfaces (some checking done by SEGMENTER, but not for all types)

— Hidden information (to keep information within the module — problems with stack-structure, file-access)

— More possibilities to restrict access on data, procedures and files

— Comfortable to handle (library-problem)

Example: Own Data in SL-Routines

PROBLEM: The principle of hidden information requires that local data is not deleted between two procedure calls. This causes problems when procedure has to be put into a SL.

WHAT WE WANT: A module which stores local data into an extra data segment before exit and refreshes the data after call.

SPECIFICATION FOR MODULE "OWN DATA"

```
PROCEDURE INITDATA (BUFFER, LENGTH);
  INTEGER ARRAY BUFFER;
  VALUE LENGTH; INTEGER LENGTH;
  OPTION EXTERNAL;

  BEGIN
    IF `first time used'
      THEN `initialize BUFFER with 0'
      ELSE `refresh BUFFER with data
              stored in data segment';
  END;
```

```
PROCEDURE UPDATEDATA (BUFFER, LENGTH);
  INTEGER ARRAY BUFFER;
  VALUE LENGTH; INTEGER LENGTH;
  OPTION EXTERNAL;

  BEGIN
    `copy contents of BUFFER into
     data segment';
  END
```

Solution No. 1

```
PROCEDURE INITDATA (BUFFER, LENGTH);
  INTEGER ARRAY BUFFER;
  VALUE LENGTH; INTEGER LENGTH;

  BEGIN
    `allocate data segment';
    IF `data segment already exists'
      THEN `copy contents into BUFFER'
      ELSE `initialize BUFFER with 0';
  END;
```

```
PROCEDURE UPDATEDATA (BUFFER, LENGTH);
  INTEGER ARRAY BUFFER;
  VALUE LENGTH; INTEGER LENGTH;

  BEGIN
    `allocate data segment';
    `copy contents of BUFFER into
     data segment';
  END;
```

But

— Extra data segment has to be "global."

Therefore:

— Other users of module "OWN DATA" will use the same data segment

— Data segment is not automatically deallocated when program terminates. So no initialization will happen after the module has been used once.

Solution No. 2

```
PROCEDURE INITIALIZEDATE;
  OPTION PRELUDE;

  BEGIN
    `allocate extra data segment';
    `mark user within data segment';
    `initialize info part';
  END;
```

```
PROCEDURE INITDATA (BUFFER, LENGTH);
  INTEGER ARRAY BUFFER;
  VALUE LENGTH; INTEGER LENGTH;

  BEGIN
    `allocate extra data segment';
    IF `used first time (info part)'
       THEN
         BEGIN
           `initialize BUFFER with 0';
           `change info part';
         END
       ELSE `copy contents into BUFFER';
  END;


PROCEDURE UPDATEDATA (BUFFER, LENGTH);
  INTEGER ARRAY BUFFER;
  VALUE LENGTH; INTEGER LENGTH;

  BEGIN
    `allocate extra data segment';
    `copy contents of BUFFER into
    data segment';
  END;


PROCEDURE FREEDATA;
  OPTION POSTLUDE;

  BEGIN
    `allocate extra data segment';
    `delete module user from info
     part';
    `free extra data segment';
  END;
```

IMAGE/COBOL: Practical Guidelines

David J. Greer
Robelle Consulting Ltd.
Aldergrove, B.C., Canada

SUMMARY

This document presents a set of practical "rules" for designing, accessing, and maintaining IMAGE databases in the COBOL environment. This document is designed to aid systems analysts, especially ones who are new to the HP3000, in producing "good" IMAGE database designs. Each "rule" is demonstrated with examples and instructions for applying it. Attention is paid to those details that make using the database trouble-free for the COBOL programmer, and maintaining the database easier for the database administrator.

CONTENTS

DATABASE DESIGN

IMAGE/3000 is the database system supplied by Hewlett-Packard;[6] it is used to store and retrieve application information. A database does not suddenly appear out of thin air; it develops through a long and involved process. At some time, a logical database design must be translated into the actual schema that implements a physical IMAGE database. This phase is the most difficult of the database development cycle.[7] The IMAGE/3000 Reference Manual[6] contains a sample database called STORE, which demonstrates most of the attributes of IMAGE. Throughout this document, the STORE database will be referenced when examples are needed.

Logical Database Design

The foundation of a new database is a logical design, which is created by examining the user requirements for input forms, for on-line enquiries, and for batch reports. The database should be viewed as an intermediate storage area for the information that comes from the input forms and is eventually displayed on the output reports.[9] [10]

Database design is normally done from the bottom up, as opposed to structured program design, which is usually done from the top down. The starting point for a database is the elements (items) that will be stored in the database. These data elements represent the user's information. In the early stages, the size and type of these elements are not needed, only the name and values.

Rule: *Start your logical database design by naming each data item, then identify what values it can have and where it will be used.*

Here is an example of a subset of data items for the STORE database:

CUST-STATUS	Two characters, attached to each customer record. Valid values are: 10=advanced, 20=current, 30=arrears and 40=inactive.
DELIV-DATE	Six numeric characters; Date, YYMMDD, attached to every sales order as the promised delivery date.
ON-HAND-QTY	Seven numeric characters, attached to every inventory record to show the current quantity of an inventory item available for shipping.
PRODUCT-PRICE	Eight numeric characters, (6 whole digits, 2 decimal places), attached to every sales record. This is the price of a product sold, on the date that the sale was made.

As the logical database design develops to deeper levels of detail, the elements needed should eventually reach a stable list. These elements should then be combined into records by grouping logically related items together.

It is important that "repetition" be recognized early in the design. An example of this is a customer's address. The most flexible method of implementing addresses is a variable number of records associated with the customer account number. Another method is to make the address field an X-type variable (e.g., X24) repeated 5 times (e.g., 5x24). Repeated items are often the most natural way to represent the user's data, so the use of repeated items is encouraged.

After the records are designed, enquiry paths must be assigned. During the early stages of database design, it is important to use elements that are readable and easy to implement with the tools at hand. This permits testing of the database using tools such as QUERY, AQ, and PROTOS.

Physical Database Design

After the local database is designed, the IMAGE schema must be developed. The restrictions of IMAGE must now be worked into the database design.

IMAGE requires that all items needed in the database be defined at the beginning of the schema, and a size and type must be associated with each. Initially, declare each item as type X (display); later, the data type may be altered.

Records are implemented as IMAGE datasets. Start by treating each record format as a master dataset.

Rule: *If a record is uniquely identified by a single key value, start by making it a master dataset (e.g., customer master record keyed by a unique customer number).*

The STORE database assumes that each CUST-ACCOUNT field is unique. Furthermore, there is only one customer record for each CUST-ACCOUNT. All of the information describing one customer is gathered together to result in the M-CUSTOMER dataset:

```
NAME:   M-CUSTOMER,         MANUAL (1/2);         <<PREFIX = MCS>>
ENTRY:
        CITY
       ,CREDIT-RATING
       ,CUST-ACCOUNT(1)        <<KEY FIELD>>
       ,CUST-STATUS
       ,NAME-FIRST
       ,NAME-LAST
       ,STATE-CODE
       ,STREET-ADDRESS
       ,ZIP-CODE
       ;
CAPACITY:  211;      <<M-CUSTOMER,PRIME; ESTIMATED>>
```

Rule: *If a "natural" master dataset will require on-line retrieval via an alternate key, drop it down to a detail dataset.*

The detail dataset will have two keys: the key field of the original master dataset, and the alternate key. You will have to create a new automatic master for the original key field, and you may have to create an automatic master for the alternate key (unless you already have a manual master dataset for that item).

Take the M-CUSTOMER dataset as an example. Assume that in addition to needing on-line access by CUST-ACCOUNT, it is also necessary to have on-line access by NAME-LAST. The following dataset structure would result:

```
NAME:   A-CUSTOMER,         AUTOMATIC (1/2),     <<PREFIX = ACS>>
ENTRY:
        CUST-ACCOUNT(2)        <<KEY FIELD>>
       ;
CAPACITY:  211;      <<A-CUSTOMER,PRIME; ESTIMATED>>

NAME:   A-NAME-LAST,        AUTOMATIC (1/2),     <<PREFIX = ANL>>
ENTRY:
        NAME-LAST(1)       <<KEY FIELD>>
       ;
CAPACITY:  211;      <<A-NAME-LAST,PRIME; CAP(A-CUSTOMER)>>

NAME:   D-CUSTOMER,         DETAIL (1/2);        <<PREFIX = DCS>>
```

```
ENTRY:
      CITY
     ,CREDIT-RATING
     ,CUST-ACCOUNT(!A-CUSTOMER)     <<KEY FIELD, PRIMARY>>
     ,CUST-STATUS
     ,NAME-FIRST
     ,NAME-LAST(A-NAME-LAST)        <<KEY FIELD>>
     ,STATE-CODE
     ,STREET-ADDRESS
     ,ZIP-CODE
     ;
CAPACITY:  210;     <<D-CUSTOMER; CAP(A-CUSTOMER)>>
```

Rule: *If an entry can occur several times for the primary key value, store it in a detail dataset.*

Detail datasets are for repetition and multiple keys. Master datasets can only contain one entry per unique key value. An example of repetition in a detail dataset is a customer address field. The customer address can be stored as a repeated field in a master dataset, but eventually there will be an address that will not fit into the fixed-size repeated field. Instead of a repeated field, use a detail dataset to store multiple lines of an address. For example:

```
ADDRESS-LINE,          X24;     << An individual line of address.  This
                                   item is used in D-ADDRESS to provide an
                                   arbitrary number of address lines for
                                   each customer.
                                >>
CUST-ACCOUNT,          Z8;      << Customer account number.  This field
                                   is used as a key to the M-CUSTOMER
```

```
                                   and D-ADDRESS datasets.
                                >>
LINE-NO,               X2;      << Used to keep address lines in D-ADDRESS
                                   in the correct order.  This field also
                                   provides a unique way of identifying
                                   each address line for every
                                   CUSTOMER-ACCOUNT.
                                >>

NAME:  D-ADDRESS          DETAIL (1/2);       <<PREFIX = DAD>>
ENTRY:
        ADDRESS-LINE
       ,CUST-ACCOUNT(!M-CUSTOMER(LINE-NO))  <<KEY FIELD, PRIMARY PATH>>
       ,LINE-NO                             <<SORT FIELD>>
       ;
CAPACITY:  844;     <<D-ADDRESS; 4 * CAP(M-CUSTOMER)>>
```

Dataset Paths

The following definition of PATHs and CHAINs comes from Alfredo Rego:[11]

> A PATH is a relationship between a MASTER dataset and a DETAIL dataset. The master and the detail must contain a field of the same type and size as a common "bond," called the SEARCH FIELD. A path is a structural property of a database.
>
> A CHAIN, on the other hand, contains a MASTER ENTRY and its associated DETAIL ENTRIES (if any), as defined by the PATH relationship between the master and the detail for the particular DETAIL SEARCH FIELD. . . . A chain is nothing more than a collection of related entries (for instance, a bank customer would be the master entry and all of this customer's checks would be the detail entries; the "chain" would include the master AND all its details; the chain for customer number 1 would be completely different from the chain for customer number 2).

Paths provide fast access at a certain cost: adding and deleting records on the path is expensive. The more paths there are, the more expensive it gets.[1] Another restriction of paths is that there can be a maximum of 64,000 records on a single path for a single key value. This sounds like a large number, but it can be very easy to expand a chain to this size if a key is specified for a specific, reporting summary program (e.g., billing cycle, in monthly billing transactions).

Rule: *Avoid more than two paths into a detail dataset.*

There are some instances where three paths are necessary, but these should be avoided as much as possible. Before adding a path, examine how the path is going to be used. If it is added just to make one or two batch programs easier to program, the path is not justified. The batch programs should serially read and sort the dataset, then merge the sorted dataset with any other necessary information from the database.

The date paths of the SALES dataset of the STORE database are good examples of unnecessary paths. Because the chain lengths of paths organized by date are almost always very long, such a chain is rarely allowed. Also, users are often interested in a large range of dates (such as a month, quarter or year), not just a specific day.

In order to obtain the same type of reporting by date, it is possible to do one of the following: (1) read the database every night and produce a report of all records entered every day; (2) keep a sequential file of all records added to the dataset on a particular day. This file can then be used as an index into the database.

These are not the only solutions to removing the date paths, but they indicate the kind of solutions that are possible. Because of the high volume and length of the average chain, date paths are prime candidates for removal from a database.

The following example demonstrates how the SALES dataset should have been declared:

```
<<  The D-SALES dataset gathers all of the sales records
    for each customer.  The primary on-line access is by customer,
    but it is necessary to have available the product sales
    records.  The PRODUCT-PRICE is the price at the time
    the product is ordered.  The SALES-TAX is computed based
    on the rate in effect on the DELIV-DATE.
>>

NAME:  D-SALES,          DETAIL (1/2);        <<PREFIX = DSA>>
ENTRY:
         CUST-ACCOUNT(!M-CUSTOMER)   <<KEY FIELD, PRIMARY PATH>>
        ,DELIV-DATE
        ,PRODUCT-NO(M-PRODUCT)       <<KEY FIELD>>
        ,PRODUCT-PRICE
        ,PURCH-DATE
        ,SALES-QTY
        ,SALES-TAX
        ,SALES-TOTAL
        ;
CAPACITY:  600;     <<D-SALES; 3 * CAP(M-CUSTOMER)>>
```

Rule: *Avoid sorted paths.*

Because sorted paths can require very high overhead when records are added or deleted, they should be avoided as much as possible. There are some instances when a sorted path makes the system and program design much easier, but this convenience must be traded off against the highest cost of maintaining sorted chains.

The most important criteria in evaluating sorted chains are: (1) whether the chain is needed for batch or on-line access. In batch, it is possible to read and sort the dataset, rather than relying on sorted chains. In an on-line program, this is usually not possible, so sorted chains are required. (2) How long is the average chain going to be? The longer the chain, the more expensive it is to keep sorted. If chains have fewer than 10 entries per key value on average, sorted chains can be permitted. (3) How are records being added to the dataset? If a sorted chain is present, and data is added to the dataset in sorted order, there is very little extra overhead in the sorted chain. If, on the other hand, data is added in random fashion, there is a very high cost associated with the sorted chain.[11-13]

Locking Strategy

Early in the database design, it is important to identify the locking necessary for the application. The easiest choice is to use database locking. Unless specific entries are going to be modified by many users, database locking should work. Remember: locking is only needed when updating, adding, or deleting entries from the database, not when reading entries. Never leave the database locked when interacting with the terminal user.

The next level of locking to be considered is dataset locking. This takes more programming, but provides for a more flexible locking strategy.

Rule: *Never permit MR capability to programmers;* instead, use lock descriptors (and a single call to DBLOCK) to lock all datasets needed.

For very complicated systems (e.g., an inventory system with inventory levels that must be continually updated), record locking should be used. The database design should help the application programmer by making the easiest possible locking strategy available for each program.[2]

Passwords

Most application systems go overboard in their use of database passwords. The simplest scheme to implement is a two-password system. The database is declared with one password for reading and one for writing. Each password is applied at the dataset level; and item-level passwords are not used.

Rule: *Use the simplest password scheme that does not violate the database integrity.*

The advantages to this scheme are that there are fewer passwords to remember, IMAGE is more efficient (because all security checks are done at the dataset level, instead of the data item level), and the user can still use tools such as QUERY, by being allowed the read-only password.

In sensitive applications, a separate dataset or database can be used to isolate data requiring special security. This still permits the simplest password scheme possible, with an extra level of security. The following example shows how to declare passwords for read-only access and read/write access on a dataset level:

```
PASSWORDS:        1 READER;
                  2 WRITER;
```

The declarations for the M-CUSTOMER and the D-SALES datasets contain "(½)" on the line that declares the name of the datasets. The "(½)" indicates that the READER and WRITER passwords are in effect for the whole dataset.

Early Database Testing

The early database design should allow the user or analyst to experiment on the database design with test data. User tools such as QUERY or AQ should be used to access the database. At this stage, the item types may be left approximate, so long as the user or analyst gets a chance to interact with the database design. The analyst should check that all requirements of the user can be met by the database design.

Rule: *Build your test databases early. Use an application tool to verify that the database design is correct.*

In some cases, the end user may not be able to access the database, but the database designer must go through this testing process. This examination of the database design may uncover design flaws which can be fixed easily at this early stage. After the logical database design has been roughly packaged as an actual IMAGE database and verified against the user requirements, the design should be optimized and the finishing touches added (see next section).

Very Complex Databases

IMAGE has a number of size restrictions that it imposes on the database design. For example, the number of items in a database is limited to 255, and the number of datasets in a database is limited to 99. For many applications, these limits pose no problems; but with the larger databases being designed today, it is not difficult to imagine databases which exceed these lmits. What can you do to get around this problem?

Bottom-Up Design

The design method outlined above must be extended. For small projects, it is adequate to simply group related data items into datasets, because the entire application will fit into one database. However, for large projects, another step is required: related datasets must be grouped into separate databases.

Multiple databases introduce new problems for the application programmer. These include larger programs, which result in larger data stacks, as well as problems with locking. In designing a multiple database system, it is best to minimize the number of programs that must use more than one database.

If an application decomposes into independent subunits, few programs will require more than one database. The design of the system and the database may have to be revised to increase the independence of the sub-systems.

POLISHING DATABASE DESIGN

The database designer has two main concerns in completing the database design. Will the application programs be able to access the database within the defined limits of the HP3000? Does the database take best advantage of COBOL and other tools available?[3–8–11–13]

Overall Performance

Rule: *Always make a formal estimate of on-line response times and elapsed times for batch jobs.* If the project is going to require additional hardware resources, it is better to know it before the project goes into production.

The following material is taken from *On Line System*

Design and Development,[9] with comments and examples to expand on the original. The HP3000 is able to perform approximately 30 I/Os per second. On various machines under different operating systems, it may be possible to obtain more than this. Because it is extremely difficult to obtain the theoretical maximum of 30 I/Os per second, it is best to plan for a maximum of 20 I/Os per second.

Each IMAGE procedure results in a specific amount of I/O. Before going ahead with a large application, the total I/O required for the application must be computed and compared against the maximum. This is done by estimating the I/O for each on-line function, then summing the I/Os of the functions that might reasonably occur concurrently. Also, the total elapsed time for batch jobs must be estimated to ensure that they will complete in the time available.

The following gives an approximate measure of the number of I/Os necessary for each IMAGE procedure in an on-line environment:

Procedure	I/O
DBGET	1
DBFIND	1
DBLOCK	0
DBUNLOCK	0
DBUPDATE	1
DBPUT	2 + 2 * Number of keys in the dataset.
DBDELETE	2 + 2 * Number of keys in the dataset.

The figures for DBPUT and DBDELETE do not take into account sorted chains. If sorted chains are kept short, the above figures will work. If sorted chains are long, the following formula gives an approximate measure of how many I/Os are required to add records in random fashion to a sorted chain:

```
2 + 2 * number of keys + (average chain length / 2)
```

All of the above figures for the number of I/Os for each IMAGE procedure are the same in batch, with one exception. If a batch program reads a dataset serially, the I/Os required will be:

```
Serial DBGET I/Os = number of records / blocking factor
```

If the batch program also does a sort of all of the selected records from the serial DBGET, the number of I/Os will be increased.

Batch Calculation Example

The following example computes how long a specific batch program will take to run; the program makes the following IMAGE calls:

```
125,000 DBGETs serial; blocking factor is 5.
 80,000 DBPUTs to a detail dataset with two keys.
 80,000 DBFINDs.
 80,000 DBGETs to the dataset with the DBFIND.
 80,000 DBUPDATEs.

 Total I/Os required =

 I/Os for DBGET  (205,000 / 5) plus
 I/Os for DBFIND ( 80,000 X 1) plus
 I/Os for DBPUT  ( 80,000 X 6) plus
 I/Os for DBUPDATE(80,000 X 1).

 equals   681,000 I/Os.

 We can do approximately 20 I/Os a second so

       681,000
       -------    =  34,050 seconds  =  9.5 hours
         20
```

If the batch program also is intended to run overnight, but is unlikely to finish in one evening, because time is also needed for backup and other daily functions.

Improving Performance

How can the total time of this program example be reduced to 3.9 hours? One way is to replace the DBPUT with a DBUPDATE. In many instances it is possible, through changes in the application and database design, to use a DBUPDATE instead of a DBPUT. This is especially true in environments where there are recurring monthly charges, which change only slowly over time.

There is another advantage to using DBUPDATE. For each DBPUT, a record is added to the database, and this record must later be deleted using DBDELETE. Because it takes as long to delete the record as it did to add it in the first place, the DBUPDATE can provide as much as an eight-fold decrease in running time, compared with DBPUT/DBDELETE.

COBOL Compatibility

When designing a database, keep in mind how the database is going to be used (COBOL, QUERY, AQ, PROTOS, etc.). The following rules apply to item types and should be used throughout the database.

Numeric Fields

When the database was first designed, all fields were initially declared as type X (display). By now you should know the likely maximum value for each data item. Once the size of each data item is fixed, the time has come to specify a more efficient data type for numeric fields.

The type of field used for numeric values depends on the maximum size of the number to be stored (i.e., the number of digits, ignoring the sign). The following table should be used in determining numeric types:

```
Number of Digits  IMAGE Data type

       <5                J1
       <10               J2
       >=10            Packed-decimal of the appropiate size.
```

Rule: *For numeric fields, use J1 for fewer than five digits; use J2 for fewer than ten digits; otherwise, use a P-field (packed-decimal) of the appropriate size.*

In COBOL, an S9(2)V9(2) COMP variable is considered to have a size of 4, or J1. The one exception to this rule is sort fields. All sort fields must be type X. If a numeric sort field is required, it must be declared as type X and redefined as zoned in all COBOL programs. Remember that packed fields in IMAGE are always declared one digit larger than the corresponding COBOL picture (S9(11) COMP-3 becomes P12) and must be allocated in multiples of four.

COBOL databases must not contain R-fields, because R-fields have no meaning in the COBOL language. Instead of an R-type field, a J-type or P-type field must be used. The STORE database contains an R-type field, CREDIT-RATING, which should have been declared as:

```
CREDIT-RATING,          J2;     << Customer credit rating.  The larger
                                   the number, the better the customer's
                                   credit.  Used to five decimal places.
                                >>
```

Key Types

Every key, whether in a master or detail dataset, must be hashed to obtain the actual data associated with the key value. Hashing is a method where a key value, such as customer number 100, is turned into an address. The method used tries to generate a different address for every key value, but in practice this is never possible. The choice of the type of key has a large bearing on how well the hashing function will work.

Rule: *Always use X-type, U-type, or Z-type keys, and never use J-type, R-type, P-type, or I-type keys.* Type X, type U, and type Z keys give the best hashing results.

When using a Z-type for a key, leave it as unsigned in all COBOL programs. Because key values rarely have negative values, there is no effect on the application by removing the sign from a zoned field. The advantages to leaving off the sign are: (1) displaying the field in COBOL or QUERY results in a more "natural" number, and (2) problems between positive, signed, and unsigned zoned numbers are avoided.

Date Fields

Rule: *Dates must be stored as J2 (S9(6) COMP) in YYMMDD format.*

This format provides the fastest access time in COBOL and takes the least amount of storage. Use a standard date-editing routine to convert from internal to external format and vice versa.[4]

The only exception to this is when a detail chain must be sorted by a date field. Because IMAGE does not allow sorting on J2 fields, X6 is used. For the chain to be sorted correctly, the date must still be stored in YYMMDD format.

Other Item Types

The only item types that should be used are J- or P-types for numeric values, and X-, U- or Z-types for keys. The K-, I- and R-types should never be used in a commercial application where COBOL is the primary development language.

Example

Earlier, in the discussion of logical database design, four items were described: CUST-STATUS, DELIV-DATE, ON-HAND-QTY, and PRODUCT-PRICE. The following example gives the actual IMAGE declaration for each of these items, according to the rules of this section:

```
CUST-STATUS,          X2;    << Defined state of a particular customer
                                account.  The valid states are:
                                10 = advance
                                20 = current
                                30 = arrears
                                40 = inactive
                             >>
DELIV-DATE,           J2;    << Promised delivery date.
                             >>
ON-HAND-QTY,          J2;    << Amount of a specific product currently
                                onhand.  Only updated upon
                                confirmation of an order.
                             >>
PRODUCT-PRICE,        J2;    << Individual product price, including
                                two decimal points.
                             >>
```

Primary Paths

Rule: *Assign a primary path to every detail dataset.*

IMAGE organizes the database so that accesses along the primary path are more efficient than along other paths. The primary path should be the path that is accessed most often in the dataset.

If there is only one path in a detail dataset, it must be the primary path. If there are two paths that are accessed equally often, but one is used mostly in on-line programs and the other mostly in batch programs, assign the primary path to the one that is used in on-line programs. A primary path is indicated by an exclamation point (!) before the dataset name that defines the path. A path with only one entry per chain should not be selected as a primary path.

The Schema

The IMAGE schema is the method by which you tell both IMAGE and the programmers what the database looks like. The schema should be designed with maximum clarity for the programmer, because IMAGE is only partly concerned with the schema's layout.

Rule: *The schema file name is always XXXXXX00, where XXXXXX is the name of the database.*

This naming convention makes locating the schema easier for all staff. The file is always located in the same group and account as the database. If the database name was STORE and the STORE database was built in the DB group of the USER account, the schema name would be STORE00.DB.USER.

Layout

A clear layout of the schema makes the programmer's job easier. Some requirements of the layout are imposed by IMAGE, but there are still a number of things that the database designer can do to make the schema more understandable.

Every database schema should start with a $CONTROL line. The $CONTROL line must always contain the TABLE and BLOCKMAX parameters. The default BLOCKMAX size of 512 should always be used when first implementing the database. Later, after careful consideration, the BLOCKMAX size may be changed. When first designing the database, $CONTROL NOROOT should be used.

The $CONTROL line should be followed by the name of the database. This is followed by a header comment. This comment describes the designer of the database, the date, the conventions used in designing the schema, abbreviations that are used within IMAGE names, and sub-systems with which the database is compatible and incompatible.

The following are the opening lines of the example STORE database:

```
$CONTROL TABLE,BLOCKMAX=512,LIST,NOROOT
BEGIN

DATA BASE   STORE;

<<                 STORE DATABASE FROM THE IMAGE MANUAL

AUTHOR:     DAVID J. GREER, ROBELLE CONSULTING LTD.

DATE:       DECEMBER 15, 1981

CONVENTIONS:

This schema is organized in alphabetic order.  All master datasets are
listed before detail datasets, and automatic masters come before
manual master datasets.

All dates are stored as J2, YYMMDD, except where they are used as
sort fields.  If a date is a sort field, it is stored as X6, YYMMDD.

The following abbreviations are used throughout the schema:

NO     = Number
CUST   = Customer
QTY    = Quantity

This database can be accessed by COBOL, QUERY, AQ and PROTOS.  Note
that the STREET-ADDRESS field is incompatible with QUERY, but AQ
can correctly add and modify the STREET-ADDRESS field.
>>
```

Naming of Items and Sets

Rule: *Names must be restricted to 15 characters; the only special character allowed in names is the dash (-).* This ensures that the names are compatible with V/3000 and COBOL.

The percent sign (%) should be replaced with the abbreviation "-PCT", and the hash sign (#) should be replaced with the abbreviation "-NO".

Item Layout

The easiest layout to implement, maintain and understand is to declare everything in the database sorted in alphabetic order. The items in the database should begin with a $PAGE command to separate the items from the header comment. Each item appears sorted by its name, regardless of the item's type or function.

In many IMAGE applications, the schema also acts as the data dictionary. For this reason, it is very important that every part of the database design be completely documented in the schema. Document each item as it is declared. To make each item stand out, the following layout should be used:

```
CUST-NO,              Z10;     << The customer number is used as a
                                  key field in the M-CUSTOMER dataset.
                                  It is also the defining path in
                                  the D-ORDER-DETAIL dataset.
                               >>
```

The item name, its type, and the comment start in the same column for every item. Each part of the item definition will stand out, and because the item names are in sorted order, the applications programmer can easily find a particular item.

Dataset Layout

Every dataset declaration must be preceeded by a header comment that describes the use of the dataset and any special facts that the programmer should be aware of.

When accessing the dataset from a COBOL program, it will be necessary to have a COBOL record which corresponds to the dataset. In order to prevent confusion between two occurrences of the same item as a field in several datasets, a prefix will be assigned to each of the variables in the COBOL buffer declaration. This prefix is selected by the database designer and must appear on the same line as the name of the database. For example:

```
<<  The M-CUSTOMER dataset gathers all of the static information
    about each customer into one dataset.  A customer must exist
    in this dataset before any sales are permitted to the
    customer.  This dataset also provides the necessary path
    into the D-SALES dataset.
>>

      NAME:   M-CUSTOMER,                  MANUAL (1/2);  <<PREFIX=MCS>>
```

The AUTOMATIC, MANUAL or DETAIL keyword must always appear in the same column. This makes reading the schema easier, and by searching the file for a string (by using \L"NAME:" in QEDIT) it is possible to produce a nice index of dataset names, their types, and their prefixes. The following example prints an index of the STORE dataset names:

```
:RUN QEDIT.PUB.ROBELLE
/LQ STORE00.DB "NAME:"
NAME:  M-CUSTOMER,        MANUAL (1/2);        <<PREFIX = MCS>>
NAME:  M-PRODUCT,         MANUAL (1/2);        <<PREFIX = MPR>>
NAME:  M-SUPPLIER,        MANUAL (1/2);        <<PREFIX = MSU>>
NAME:  D-INVENTORY,       DETAIL (1/2);        <<PREFIX = DIN>>
NAME:  D-SALES,           DETAIL (1/2);        <<PREFIX = DSA>>
```

Rule: *Automatic master datasets have names that start with "A-".*

They must be declared immediately after the item declarations, separated from item declarations by a $PAGE command, and they must appear in alphabetic order.

Rule: *Manual master datasets have names that start with "M-".*

The manual master datasets follow the automatic master datasets, again preceded by a $PAGE command. Like the automatic masters, the manual master datasets must be declared in alphabetic sequence.

Rule: *Detail dataset names start with "D-".*

The detail datasets follow the manual master datasets, and the two are separated by a $PAGE command. The detail datasets also appear in alphabetic order.

Field Layout

Without exception, the fields in every dataset must be declared sorted alphabetically. There is a strong tendency to try to declare the fields within a dataset in some other type of logical grouping. Because this logical grouping exists only in the mind of the database designer and cannot be explicitly represented in IMAGE, it should never be used. By declaring fields in sorted order, the applications programmer can work much faster with the database, since no time has to be spent searching for fields within each dataset.

The database designer can still group fields together in a dataset by starting each field with the same prefix. If a dataset contains a group of costs, they might be called VAR-COSTS, FIX-COSTS and TOT-COSTS. To group these items together in the dataset, call them COSTS-VAR, COSTS-FIX and COSTS-TOT. This maintains the sorted field order in each dataset, while allowing for logical grouping of fields.

Most datasets contain one or more key fields. A key field is specified by following it with (). Because the () pair is sometimes hard to see, a comment should be included beside every key field, indicating that the field is a key. In a detail dataset, the primary key should include a comment to that effect. The following example shows how to declare the fields in a dataset:

```
<<  The D-SALES dataset gathers all of the sales records
    for each customer.  The primary on-line access is by customer,
    but it is necessary to have available the product sales
    records.  The PRODUCT-PRICE is the price at the time
    the product is ordered.  The SALES-TAX is computed based
    on the rate in effect on the DELIV-DATE.
>>

NAME:  D-SALES,           DETAIL (1/2);       <<PREFIX = DSA>>
```

```
ENTRY:
         CUST-ACCOUNT(!M-CUSTOMER)    <<KEY FIELD, PRIMARY PATH>>
        ,DELIV-DATE
        ,PRODUCT-NO(M-PRODUCT)        <<KEY FIELD>>
        ,PRODUCT-PRICE
        ,PURCH-DATE
        ,SALES-QTY
        ,SALES-TAX
        ,SALES-TOTAL
        ;
CAPACITY:  600;     <<D-SALES; 3 * CAP(M-CUSTOMER)>>
```

Capacities

Analysis of the data flow of the application should result in an approximate capacity for each dataset.

Rule: *The capacity of master datasets must be a prime number.*

To see if a number is prime :RUN the PRIME program contributed by Alfredo Rego. Master datasets should never be more than 80% full (see DBLOADNG below, under "Database Maintainence"), and detail datasets should never be more than 90% full.

The line with the capacity must be formatted in the following way:

```
CAPACITY:   211;     <<M-CUSTOMER,PRIME; ESTIMATED>>
```

The comment after the capacity gives a method for determining the approximate capacity of the dataset. Most detail datasets have a capacity that is related to the master datasets having paths into the detail datasets. These relationships should be described in the capacity comment.

By doing a \L"CAPACITY", it is possible to obtain quickly an index of the capacity of each dataset in the schema. Because the capacity is always the last line of each dataset declaration, doing a \L"M-CUSTOMER" will identify the beginning and ending declarations for the M-CUSTOMER dataset. The following example lists the capacity of the datasets in the STORE database:

```
:RUN QEDIT.PUB.ROBELLE
/LQ STORE00.DB "CAPACITY:"
CAPACITY:  211;     <<M-CUSTOMER,PRIME; ESTIMATED>>
CAPACITY:  307;     <<M-PRODUCT,PRIME; ESTIMATED>>
CAPACITY:  211;     <<M-SUPPLIER,PRIME; ESTIMATED>>
CAPACITY:  450;     <<D-INVENTORY; 2 * CAP(M-SUPPLIER)>>
CAPACITY:  600;     <<D-SALES; 3 * CAP(M-CUSTOMER)>>
```

Final Checkout

After the schema is entered into a file, it must be :RUN through the schema processor, and any typing mistakes should be eliminated:

```
:FILE DBSTEXT=STORE00.DB
:FILE DBSLIST;DEV=LP;CCTL
:RUN DBSCHEMA.PUB.SYS;PARM=3
```

The table produced at the end of the schema should be studied. The following anomalies should be checked:

1. Large-capacity master datasets with a blocking factor less than four (either increase the BLOCKMAX size to 1024, or change the master dataset to a detail dataset with an automatic master dataset).
2. The blocksize is too small (IMAGE optimizes the blocking factor to minimize disc space); use REBLOCK of ADAGER to increase the blocking factor. The blocksize of all dataset blocks should be as close to the BLOCKMAX size as possible.
3. Are there more than two paths into a detail dataset? If there are, can some of them be deleted?

Establishing the Programming Context

By using IMAGE, the COBOL programmer's job should be simplified, since all access to the database is done through the well-defined IMAGE procedures. Like most powerful tools, IMAGE (and COBOL) can be abused by the unsuspecting user.

Rule: *Define a standard IMAGE communication area* *and put this area in the COPYLIB.*

The starting point for using IMAGE is the standard parameter area, which includes the IMAGE status area, the various access modes used, a variable for the database password, and a number of utility variables which are needed when using IMAGE. For example:

```
05  DB-ALL-LIST                 PIC X(2) VALUE "@ ".
05  DB-SAME-LIST                PIC X(2) VALUE "* ".
05  DB-NULL-LIST                PIC S9(4) COMP VALUE 0.
05  DB-DUMMY-ARG                PIC S9(4).
05  DB-PASSWORD                 PIC X(8).
05  DB-MODE1                    PIC S9(4) COMP VALUE 1.
05  DB-KEYED-READ               PIC S9(4) COMP VALUE 7.
05  DB-STATUS-AREA.
    10  DB-COND-WORD            PIC S9(4) COMP.
        88  DB-STAT-OK              VALUE ZEROS.
        88  DB-END-OF-CHAIN         VALUE 15.
        88  DB-BEGIN-OF-CHAIN       VALUE 14.
        88  DB-NO-ENTRY             VALUE 17.
        88  DB-END-FILE             VALUE 11.
        88  DB-BEGIN-FILE           VALUE 10.
    10  DB-STAT2                PIC S9(4) COMP.
    10  DB-STAT3-4              PIC S9(9) COMP.
    10  DB-CHAIN-LENGTH         PIC S9(9) COMP.
        88  DB-EMPTY-CHAIN      VALUE ZEROS.
    10  DB-STAT7-8              PIC S9(9) COMP.
    10  DB-STAT9-10             PIC S9(9) COMP.
```

Rule: *Establish naming standards for all variables associated with IMAGE databases.*

Standard prefixes must be used on all database variables, including the database, dataset, data field and dataset buffer declarations. A suggestion is to start all database variables with "DB-", all dataset names with "DB-SET-", and all database buffer declarations with "DB-BUFFER-". Data field names are prefixed by the special dataset prefix (which the designer established in the schema), so that each field has a unique name. For example:

```
01  DATASET-M-PRODUCT.

    05  DB-SET-M-PRODUCT                PIC X(10) VALUE "M-PRODUCT;".

    05  DB-BUFFER-M-PRODUCT.
        10  MPR-PRODUCT-DESC            PIC X(20).
        10  MPR-PRODUCT-NO              PIC 9(8).
```

Field Lists

The selection of the type of field lists depends on the answer to this question: Can your total application be recompiled in a weekend?

Rule: *Use "@" field list is you can recompile in a weekend (prepare a COPYLIB member for each dataset); use "*" field list otherwise and hire a DBA!*

If the answer to the question is "yes," the at ("@") field list and full buffer declarations should be used when accessing the database. This method requires that all dataset buffers be declared and added to the COPYLIB. If a dataset changes, the buffer declaration must be changed in the COPYLIB, and all affected programs must be recompiled. The simplest solution is to recompile the complete application system whenever a dataset changes.

There must be two complete COPYLIBs available for every application. One is for production, and one is for development.

Rule: *Use a test COPYLIB during development. Double-check that all existing programs will recompile and :RUN correctly before moving the new COPYLIB into production!*

When a database is restructured, the buffer declarations are first changed in the development COPYLIB. When the new database is put into production, the development COPYLIB is also moved into production, as well as any programs that required modification or recompilation.

If the application system is so large that it cannot be recompiled in a weekend, it should use partial field lists and the same ("*") field list. This requires that an application program declare a matching field list and buffer area for each dataset that it accesses. The field list declares the minimum subset of the dataset that the application program needs.

Because partial field lists are more expensive at run time, the applications programmer must code a one-time call to DBGET for every dataset that the application program will use. The same ("*") field list is used on all subsequent DBGET calls. Note that this can cause problems if a common subroutine is called that uses one of the same datasets, but with a different field list.

In order to maintain an application with partial field lists, there must be a way to cross reference every program/dataset relationship. When a dataset changes, the cross reference system is checked to see which programs use the dataset. Each of these programs must be examined to see if it is affected by the change to the dataset. It is not enough to fix the COPYLIB and recompile, since the field declarations are in the individual source files, not in the COPYLIB file.

Dataset Buffers

The database designer assigns a short, unique prefix to each dataset of each database. These prefixes are used in the declaration of the database buffers for the datasets. In addition, dataset buffer declarations must include all 88-level definitions for flags, and subdefinitions for IMAGE fields that are logically subdivided within the application.

The following is the full buffer declaration for the M-CUSTOMER dataset of the STORE database. Note that each variable is prefixed with "MCS-", which is the prefix that was assigned by the database designer.

```
01  DB-BUFFER-M-CUSTOMER.
    05  MCS-CITY                     PIC X(12).
    05  MCS-CREDIT-RATING            PIC S9(4)V9(5) COMP.
    05  MCS-CUST-ACCOUNT             PIC 9(10).
    05  MCS-CUST-STATUS              PIC X(2).
        88  MCS-CUST-ADVANCE         VALUE "10".
        88  MCS-CUST-CURRENT         VALUE "20".
        88  MCS-CUST-ARREARS         VALUE "30".
        88  MCS-CUST-INACTIVE        VALUE "40".
    05  MCS-NAME-FIRST               PIC X(10).
    05  MCS-NAME-LAST                PIC X(16).
    05  MCS-STATE-CODE               PIC X(2).
    05  MCS-STREET-ADDRESS           PIC X(25) OCCURS 2.
    05  MCS-ZIP-CODE.
        10  MCS-ZIP-CODE-1           PIC X(3).
        10  MCS-ZIP-CODE-2           PIC X(3).
```

Repeated items should be declared with an occurs clause, or sub-divided, whichever the application requires. For example, a cost field may be declared as a repeated item representing fixed, variable, overhead, and labor costs. Rather than declare the costs field as a repeated item in the actual buffer declaration, subdivide it into the four costs. For example, assume a declaration for costs such as:

```
COSTS,          4J2;    <<Cost of an item.  Each cost has two
                          decimal points and the cost item
                          is broken down as follows:
                          COSTS(1) = Variable costs
                          COSTS(2) = Fixed costs
                          COSTS(3) = Overhead costs
                          COSTS(4) = Labour costs
                        >>
```

Assuming that the COSTS field was declared in the D-INVENTORY dataset, which has a prefix of "DIN", the following buffer declaration would be used for the COSTS field:

```
01  DB-BUFFER-D-INVENTORY.
    05  DIN-COSTS.
        10  DIN-VARIABLE-COSTS       PIC S9(7)V9(2) COMP.
        10  DIN-FIXED-COSTS          PIC S9(7)V9(2) COMP.
```

```
        10  DIN-OVERHEAD-COSTS       PIC S9(7)V9(2) COMP.
        10  DIN-LABOUR-COSTS         PIC S9(7)V9(2) COMP.
```

Rule: *Prepare sample COBOL calls to IMAGE in source files, with one IMAGE call per file.*

The sample IMAGE calls should be organized with one parameter per line. When programming, these template IMAGE calls must be copied into the COBOL program and modified with the database name, dataset name, and any other necessary parameters.

General purpose SECTIONS, declared in the COPYLIB, should NOT be used for the IMAGE calls. These SECTIONS obscure the meaning of the COBOL code. In addition, they can cause unnecessary branches across segment boundaries.

A scheme for handling fatal IMAGE errors must be declared, and the sample IMPAGE calls should refer to the fatal-error section. Here is a sample call to the IMAGE routine DBFIND:

```
        CALL "DBFIND" USING DB-
                            DB-SET-
                            DB-MODE1
                            DB-STATUS-AREA
                            DB-KEY-
                            DB-ARG-
         IF NOT DB-STAT-OK AND NOT DB-NO-ENTRY THEN
             PERFORM 99-FATAL-ERROR.
```

The fatal-error section (99-FATAL-ERROR) should call DBEXPLAIN. It should also cause the program to abort, and the system job-control word should be set to a fatal state. Note that just using STOP RUN will not set the system job-control word to a fatal state. The following is an example of a fatal-error section. The routine MISQUIT calls the QUIT intrinsic, which causes the program to abort.

```
$PAGE "[99]  FATAL ERROR"
**************************************************************
*  THIS SECTION DOES THE FOLLOWING:                          *
*  1.  CALLS DBEXPLAIN WITH THE IMAGE STATUS AREA.           *
*  2.  CALLS MISQUIT TO ABORT THE PROGRAM.                   *
*                                                            *
*  NOTE:  THIS MODULE MUST ONLY BE CALLED AFTER A FATAL ERROR*
*         HAS OCCURED WHEN CALLING AN IMAGE ROUTINE.         *
*                                                            *
**************************************************************

 99-FATAL-ERROR                SECTION.

     CALL "DBEXPLAIN" USING DB-STATUS-AREA.

     CALL "MISQUIT" USING DB-COND-WORD.

 99-FATAL-ERROR-EXIT.  EXIT.
```

Rule: *Avoid tricky data structures,* especially if they cannot be easily retrieved and displayed with the available tools (QUERY, AQ, PROTOS, QUIZ, etc.).

Some examples of data structures to avoid: (1) julian dates; (2) bit maps; (3) alternate record structures (REDEFINES); (4) implied and composite keys/paths; and (5) implied description structures. The more complicated the database structure, the more likely it is that programming or system errors will be created as a result of the database design.

Database Maintenance

There are a number of steps that the database administrator must take in order to guarantee that a database remains clean after it is implemented. A number of standard programs must be run against each production database at least once a month; others must be run daily.

Backup

A number of other people have commented on the backup problem of databases,[12] but the problem is important enough to deserve comment again. Most HP3000 shops do a full backup once a week and a partial backup once a day. This is normally sufficient for most purposes (e.g., source files, PUB.SYS, utilities), but it is not adequate for most IMAGE applications. An IMAGE database consists of several interrelated files. A database that is missing one dataset is nearly useless.

Rule: *EVERY backup tape should include ALL of the* *files of ALL of the database that are used in day-to-day applications.*

There should be an easy way to store complete databases onto partial backup tapes, without having to do selective stores. The BACKUP program (available from the San Antonio Swap Tape) helps solve this problem. The BACKUP program is run once a day against every production database. It accepts the database name as input and causes the last-modified date to be changed to today's date on every file of the database. This causes the entire database to be included on the daily partial backup.

In addition, the BACKUP program prints a listing with the following information: the dataset name, the current number of entries in the dataset, and the capacity of the dataset. Further, the BACKUP program examines the relationship between the number of entries and the capacity of each dataset, and prints a warning if it thinks the capacity is too small. This listing must be checked daily, in order to have time to expand the capacity of a dataset before it is exceeded.

Measuring Database Performance (DBLOADNG)

The performance of a given database will change as the database matures.

Rule: *The performance of every application database should be measured at least once a month.*

There is one program that will measure, in great detail, the performance of an IMAGE database. This program is DBLOADNG,[1-12] and it is available from the HPIUG contributed library.

DBLOADNG examines the performance of both master and detail datasets, and reports a large number of statistics. The most important are the percentage of secondaries in master datasets, and the elongation of detail datasets.

If there are a large number of secondaries in master datasets, either the hashing algorithm is not working well, or the capacity of the dataset needs to be increased. Note that the hashing performance of a key, such as customer number, can be improved by adding a check digit to every customer number.

The "elongation" of a detail dataset indicates whether logically related records are being stored physically adjacent. For primary paths, the elongation factor should be very small (1=perfect), since IMAGE tries to place records of a primary-path in the same disc block (see the DBLOADNG documentation and *Optimizing IMAGE: An Introduction.*[1]

If the performance of detail datasets is very poor because logically related records have been spread around the disc, there is only one solution: RELOAD the database using DBUNLOAD/DBLOAD. This will cause the detail dataset to be organized along the primary path, and could result in significant performance improvements.

Logical Database Maintenance

During the design phase of an IMAGE database, many logical assumptions are made about the data in the database. Some assumptions might be: (1) status fields, which are two characters long in a detail dataset, but have a long description in a master dataset; (2) keys that are stored in detail datasets, but do not have an explicit path into a master dataset; and (3) IMAGE chains that are limited to a specific length (e.g., one address per customer) or a range of lengths (e.g., no more than 10 items per order).

Rule: *When designing a database, keep a list of logical assumptions.*

These assumptions are dangerous, because they must be maintained by the application software, not by IMAGE.

Rule: *A program to check logical assumptions should be implemented for every application system.*

This program is often called DBREPORT, and its purpose is to check these logical assumptions. DBREPORT is often left until last, and often never implemented. This is unfortunate, since the DBREPORT program is *the* most important program in an application system.

In Alfredo Rego's paper, *DATABASE THERAPY: A practitioner's experiences*[12], he describes periodic checkups for a database. The following is taken from his paper:

> Please notice that a good diagnosis system must be nasty and sadistic by nature. It has as its primary objective to FIND ERRORS, not to certify a system as being error-free (there is no such system anyway!). A good diagnosis system must also be extremely patient and humble, since it will fail many times. Please keep in mind that there is a psychological inversion in effect here: A good diagnosis system fails if it does not detect any errors. And most of the time it will not detect any errors, since we hope and assume that the entity being tested is reasonably error-free."[12]

The DBREPORT program must be designed with Alfredo's philosophy in mind. It should check EVERY dataset in an application, and it should check EVERY record for logical consistency. This includes simple checks to see that every field in every dataset is within a reasonable limit. Examples of this are status fields that take on values from 1 to 10, but which are implemented as J1. A J1 variable can take on values from −32768 to +32767, which is certainly a larger range than 1 to 10.

The DBREPORT program must check all logical dataset relationships. What happens if every customer record has its address in a detail dataset? If the system crashes while the user is adding a new customer, the address record may not be added. DBREPORT must check for these types of relationships (what will your billing program do when it can't find an address?).

ADAGER

Rule: *If an application system is going to depend on IMAGE, ADAGER is a requirement, not an option.*

ADAGER provides all of the restructuring facilities necessary to maintain IMAGE databases; these transformations cannot be accoplished with DBLOAD/DBUNLOAD. Without ADAGER, numerous conversion programs must be written.

While DBLOAD/DBUNLOAD can be used for some simple database restructuring, it is prone to err. ADAGER is designed to be friendly to the end user, but, more importantly, ADAGER guides the user through every phase of the database restructuring process.

ADAGER provides a powerful facility, but it can also be misused by the unsuspecting. In order to make ADAGER changes effectively, test them first on a development database. Following changes to the database structure, the application programs must be recompiled (with buffers changed in the development COPYLIB), and each program must be tested against the new database.

Currently, ADAGER cannot be run from batch (at least, not conveniently), nor does it produce a hard-copy audit trail of the changes to a database.

Rule: *ADAGER must be run on a printing terminal.*

Keep the listing of the ADAGER changes to the test database. Use it to verify that the changes to the production database match exactly the changes to the test database. After changing the production database, move the development COPYLIB into production and recompile all affected programs. File the hard-copy listing of the ADAGER changes and keep it for future reference.

Because the schema is also used as the data dictionary, it must be modified to indicate the new database design. ADAGER's SCHEMA function can be used to double check that all schema changes were made properly. When modifying the database schema, be sure to apply all of the rules in the Schema section of this paper.

BIBLIOGRAPHY

To gain a complete understanding of IMAGE, study the references in this bibliography. A suggested order of study is: References 6, 7, 9, 10 and 11 for more ideas on database design; 5 for some hints on common programming errors; and 1, 3, 8, 12 and 13 for notes on optimizing IMAGE databases and application systems in general. Reference 1 is an excellent introduction to database optimization, and it includes a discussion of the DBLOADNG program.

[1]Rick Bergquist, *Optimizing IMAGE: An Introduction,* HPGSUG 1980 San Jose Proceedings.

[2]Gerald W. Davidson, *Image Locking and Application Design,* Journal of the HPGSUG, Vol. IV, No. 1.

[3]Robert M. Green, *Optimizing On-Line Programs,* Technical Report, second edition, Robelle Consulting Ltd.

[4]Robert M. Green, *SPLAIDS2 Software Package,* contains date editing routine (SUPRDATE) available from Robelle Consulting Ltd.

[5]Robert M. Green, *Common Programming Errors With IMAGE/3000,* Journal of the HPGSUG, Vol. I, No. 4.

[6]Hewlett-Packard, IMAGE/3000 Reference Manual.

[7]Karl H. Kiefer, *Data Base Design – Polishing Your Image,* HPGSUG 1981 Orlando Proceedings.

[8]Jim Kramer, *Saving the Precious Resource – Disc Accesses,* HPGSUG 1981 Orlando Proceedings.

[9]Ken Lessey, *On Line System Design and Development,* HPGSUG 1981 Orlando Proceedings.

[10]Brian Mullen, *Hiding Data Structures in Program Modules,* HPGSUG 1980 San Jose Proceedings.

[11]Alfredo Rego, *Design and Maintenance Criteria for IMAGE/3000,* Journal of the HPGSUG, Vol. III, No. 4.

[12]Alfredo Rego, *DATABASE THERAPY: A practitioner's experiences,* HPGSUG 1981 Orlando Proceedings.

[13]Bernadette Reiter, *Performance Optimization for IMAGE,* HPGSUG 1980 San Jose Proceedings.

Using COBOL, VIEW and IMAGE A Practical Structured Interface for the Programmer

Peter Somers
Cape Data, Inc.

INTRODUCTION

VIEW or V/3000, Hewlett-Packard's screen handler offers a convenient and versatile method of data collection. To fully utilize the capabilities of VIEW requires the application programmer to go beyond the routines available using the ENTRY program. Ideally the data entry routine will include complete editing including IMAGE data base checking and comprehensive error messages. The routine should allow the programmer to quickly "plug in" new applications and easily perform maintenance. Additionally the program will provide utility routines for data confirmation, screen refreshing, paging, etc.

At our shop, Cape Data, we developed a general purpose VIEW and IMAGE interface program. This program written in structured COBOL allows new applications to go up, with custom editing, in a fraction of the time previously required. The following discussion will cover this interface routine and its application. I will assume that the user has basic knowledge of both VIEW and COBOL.

TABLE OF CONTENTS

1. Screen Design

When designing your VIEW input screens using FORMSPEC, the following techniques will help you get the most out of VIEW.

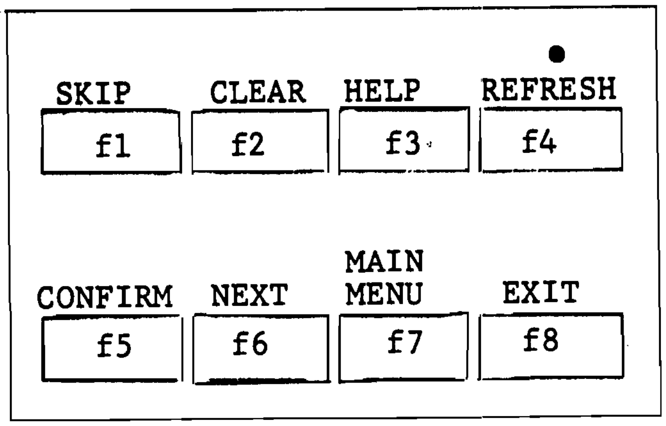
A. *Error Message Fields:* Add error message fields during form design wherever needed. Place the error message field next to or under the corresponding data field. The error message fields will remain invisible unless your program writes a message to the field. Create the field with an enhancement of B (blink) and a field type of D (display) and an initial value of spaces (Fig. 1, Field 3). The last 24th line of the screen is reserved for program error messages.

B. *VIEW Editing:* As a general rule let VIEW do as much editing as possible. On numeric fields let VIEW zero fill and test for numeric input during the FIELD portion of VIEW's editing. On alpha-numeric input fields allow VIEW to left justify the data and optionally upshift lower case characters (Fig. 1, Field 2).

C. *Title:* We reserve a Title area on all forms using Field #1. The title field is initialized by VIEW to a save field value. The VIEW forms file can contain the title and any other constants in SAVE FIELDS.

Function Keys

When using a formatted screen program with a Hewlett-Packard 2640-2645 type terminal, a special set of function keys are used by the programs. The 8 function keys are located on the upper right hand side of the terminal's keyboard. They are labeled with blue letters *f1* through *f8* in 2 rows. A blank Hewlett-Packard template labels the function keys (#7120-5525). On the 2620 family of terminals, the keys are labeled programmatically.



Function Keys

The function keys are used as follows in the interface program: *f1* SKIP — This key will cause the cursor to skip to the next block of data. This is useful if you have a number of fields to skip. The tab key only skips a field at a time where the SKIP key will skip to the next block of data.

f2 CLEAR (RESET) — This key causes the screen to clear all fields and set the initial field values (usually blanks). This key is useful if you have created a mess on the screen and want to start over again.

f3 HELP — This key will cause the program to display an instruction screen. A special set of instructions can be displayed relating to the particular form on the screen when the HELP key was pushed. When you have finished reading the HELP instructions, push the ENTER key to return to the last form or the MENU (f7) KEY to return to the MAIN MENU.

f4 REFRESH — This key resets the terminal, erases the screen and brings up a fresh copy of the form. If you loose your form due to a power or line failure or the terminal hangs up, the REFRESH KEY will restore the terminal to normal operation.

f5 CONFIRM — This key is used when you have changed a record in the EDIT mode, or if the program wants confirmation that the data on the screen is acceptable. The program will prompt with a message at the bottom of the screen when a CONFIRM is desired. Before a CONFIRM is requested the data must pass all normal program edits.

Note:

The terminal normally does not read data when the function keys are pushed. If you need to read the screen contents after a function key has been pushed call the IMMVREADFIELD Subroutine to force a read.

f6 NEXT — This key will cause the program to go to the NEXT form or next step.

f7 MAIN MENU — This key causes the program to display the MAIN MENU SELECTION form. Use this key to change from one program mode to another.

f8 EXIT — This key ends the program.

3. COBOL Application Program

A. *General:* The attached COBOL program has sufficient structure to allow the programmer to readily plug in applications without having to spend additional time coding VIEW procedures. At Cape Data, I have used this program layout to do extensive data entry routines which edit against the IMAGE data base, and provide detailed error messages and help routines. The program procedure division consists of 3 parts:

1. Opening
2. Main Loop
3. Closing

The program contains routines which call the VIEW procedures listed in Fig. 7. The program also contains special VIEW data fields in working storage.

1. Opening — The program opens the terminal, forms file, the data base and displays the MENU screen.

2. Main Loop — After opening the program performs the Main Loop until either the *f8* (exit) function key is pushed, or the program encounters a fatal error. The Main Loop consists of 3 parts:

a. Read Keys and Screen
b. Edit Input
c. Process Input (if valid) and Refresh Screen

3. Closing — The program closes the terminal file, forms file and data base.

B. *Program Data Division Considerations:* The working storage area contains the buffers needed by the various VIEW procedures used. Every VIEW procedure called uses the VIEW-COM buffer (Fig. 2). Most of the fields useful to the programmer have self-explanatory names. Remember the V-Language field must be set to zero for a COBOL program.

The data area passed between the program and VIEW (using VGETBUFFER and VPUTBUFFER calls) is defined as DATA-BUF (Fig. 3). This Buffer is redefined for each screen layout. Note the title field and error message fields.

The program uses a forms table which contains the MENU selection character, form number, next form number, and help form number for each routine. When the user makes a selection from the Screen Menu, the program scans this table to find the corresponding screen references (Fig. 4). The program picks up the Form Name corresponding to the Screen Number from the Form Name Table (Fig. 6). Additional routines and forms can quickly be added by making additional entries in the tables.

3. Program Main Loop

The program goes to the MAIN LOOP and remains there until the user pushes the *f8* (exit) function key. The program performs a terminal read (VREADFIELDS) each time either a function key or the enter key is depressed. The program then tests to see if any function keys were pushed (VIEW returns the key number pushed into the last key field of the VIEW-COM buffer).

If the ENTER key was pushed (key zero) the program will perform the edit routine corresponding to the screen routine selected. Within each edit routine the program does the following steps (Fig. 5):

1. Zeros the field error array and set the field count.
2. Performs VIEW edits (VFIELDEDITS).
3. Get the data buffer from VIEW (VGETBUFFER)
4. Clear the error message fields.
5. Performs any user defined edits (if an error is detected, a flag is set in the field error array and a message is moved to the appropriate error field)
6. The data area is sent back to VIEW (VPUTBUFFER)
7. Perform VIEW edits again (VFIELDEDITS). This zero fills and justifies data.

8. The field error array is scanned and any error fields are set to blink (VSETERROR).
9. The screen is updated and displayed (VSHOW-FORM).

If the routine passes all edits, the program then goes to the corresponding valid record routine. If an error exists or a function key was depressed, the program will do the appropriate error and screen enhancing routines.

4. SPL Forced Read Subroutine

```
$CONTROL SUBPROGRAM
<< KEPT AS IMMREAD >>
<< THIS ROUTINE IS CALLED TO FORCE              >>
<<  AN IMMEDIATE READ (RE-READ FOR DATA)        >>
<<  IN PARTICULAR CASES WHERE THE USER          >>
<<  USED A SOFT KEY TO INDICATE ACTIONS         >>
<<  HE/SHE WANTS PERFORMED WITH THE DATA        >>
<<  THAT HAS BEEN ENTERED ON THE SCREEN         >>
<< SINCE THE HITTING OF A SOFT KEY DOES         >>
<<  NOT TRANSFER THE ACTUAL BUFFER DATA         >>
<<  A CALL TO THIS ROUTINE OR ONE LIKE          >>
<<  IT IS NEEDED TO GET THE SCREEN DATA         >>
<<  INTO THE PROGRAM WHERE IT CAN BE            >>
<<  WORKED UPON                                 >>
<< IN COBOL, THE CALL WOULD BE                  >>
<<   CALL "IMMVREADFIELDS" USING VCONT          >>
<<   WITH VCONT BEING THE V/3000 CONTROL        >>
<<   AREA                                       >>
<< IN THE USER PROGRAM, THIS SHOULD BE          >>
<<  TREATED EXACTLY AS IF IT HAD BEEN A         >>
<<  CALL TO "VREADFIELDS"                       >>
BEGIN PROCEDURE IMMVREADFIELDS(CONT);
INTEGER ARRAY CONT;
BEGIN
PROCEDURE VREADFIELDS(C);
INTEGER ARRAY C;
OPTION EXTERNAL;
CONT(55).(13:2):=%1;
VREADFIELDS(CONT);
CONT(55).(13:2):=0;
END;
END.
```

```
FORMSPEC  VERSION A.00.01
FORMS FILE: BUDGFORM.DEVELOP.FDEVELOP

FORM: VENDOR_DATA
   REPEAT OPTION: N

   NEXT FORM OPTION: C
   NEXT FORM: BUDMAINT_HELP

VENDOR MASTER SPECIFICATONS, INPUT & EDITING
********* ********* ********* ********* ********* ********* ********* **********
          TITLE__________________________________________________________________
              V E N D O R - M A S T E R   I N F O R M A T I O N

              VENDOR NUMBER   V_NBR_        VEND_ERR__________________

              VENDOR'S NAME   VEND_NAME_____________________
              VENDOR ADDRESS  VEND_ADDR_____________________
                              VEND_ADDR2____________________
              VENDOR'S CITY   VEND_CITY__________  STATE: ST  ZIP: V_ZIP____


              PAYMENT ADDRESS P_ADDRESS_____________________
                              P_ADDRESS2____________________
              PAYMENT CITY    P_CITY_____________  STATE: PS  ZIP: P_ZIP____

    (PAYMENT ADDRESS USED ONLY IF YOU WANT PAYMENTS GO TO A DIFFERERNT ADDRESS)

                   VENDOR'S PHONE NUMBER: (AC_) EXC-PHON


           1099 CODE IN       VENDOR CODE VC           VENDOR STATUS VS
                         CODE_ERR__________________   STATUS_ERR________________
********* ********* ********* ********* ********* ********* ********* **********


FIELD: TITLE
   NUM: 1    LEN: 69    NAME: TITLE               ENH: NONE  FTYPE: D  DTYPE: CHAR
   INIT VALUE:
*** PROCESSING SPECIFICATIONS ***
INIT
SET TO SFTITLE


FIELD: V_NBR
   NUM: 2    LEN: 6     NAME: V_NBR               ENH: I     FTYPE: R  DTYPE: DIG
   INIT VALUE:
*** PROCESSING SPECIFICATIONS ***
FIELD
JUSTIFY RIGHT
FILL LEADING "0"


FIELD: VEND_ERR
   NUM: 3    LEN: 26    NAME: VEND_ERR            ENH: B     FTYPE: D  DTYPE: CHAR
   INIT VALUE:


FIELD: VEND_NAME
   NUM: 4    LEN: 30    NAME: VEND_NAME           ENH: I     FTYPE: R  DTYPE: CHAR
   INIT VALUE:
```

Figure 1

```
       01 VIEW-COM.
           05 V-STATUS                PIC S9(4)  COMP   VALUE ZERO.
           05 V-LANGUAGE              PIC S9(4)  COMP   VALUE ZERO.
           05 V-COM-AREA-LEN          PIC S9(4)  COMP   VALUE 60.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 VIEW-MODE               PIC S9(4)  COMP   VALUE ZERO.
           05 LAST-KEY                PIC S9(4)  COMP   VALUE ZERO.
           05 V-NUM-ERRS              PIC S9(4)  COMP   VALUE ZERO.
           05 V-WINDOW-ENH            PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 V-CFNAME                PIC X(15)  VALUE SPACES.
           05 FILLER                  PIC X      VALUE SPACES.
           05 V-NFNAME                PIC X(15)  VALUE SPACES.
           05 FILLER                  PIC X      VALUE SPACES.
           05 V-REPEAT-OPT            PIC S9(4)  COMP   VALUE ZERO.
           05 V-NF-OPT                PIC S9(4)  COMP   VALUE ZERO.
           05 V-NBR-LINES             PIC S9(4)  COMP   VALUE ZERO.
           05 V-DBUF-LEN              PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 V-DELETE-FLAG           PIC S9(4)  COMP   VALUE ZERO.
           05 V-SHOW-CONTROL          PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 V-NUM-RECS              PIC S9(6)  COMP   VALUE ZERO.
           05 V-REC-NBR               PIC S9(6)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 V-TERM-FILE-NBR         PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
           05 FILLER                  PIC S9(4)  COMP   VALUE ZERO.
```

Figure 2

```
 01  DATA-BUF.
     05 FILLER          PIC X(69).
     05 DATA-IN         PIC X(443).

 01 MENU-DATA       REDEFINES DATA-BUF.
     05 FILLER                  PIC X(69).
     05 SELECT-IN               PIC X.
     05 SELECT-ERR            PIC X(26).

 01 VEND-IN       REDEFINES DATA-BUF.
     05 FILLER                  PIC X(69).
     05 VEND-NBR                PIC X(6).
     05 VEND-NBR-ERR            PIC X(26).
     05 VEND-NAME               PIC X(30).
     05 S-ADDRESS               PIC X(30).
     05 S-ADDRESS2              PIC X(30).
     05 S-CITY                  PIC X(20).
     05 S-STATE                 PIC X(2).
     05 S-ZIP                   PIC X(10).
     05 P-ADDRESS               PIC X(30).
     05 P-ADDRESS2              PIC X(30).
     05 P-CITY                  PIC X(20).
     05 P-STATE                 PIC X(2).
     05 P-ZIP                   PIC X(10).
     05 PHONE-NBR.
      10 PHONE-AC               PIC X(3).
      10 PHONE-EX               PIC X(3).
      10 PHONE-NO               PIC X(4).
     05 FLAG-1099               PIC X(2).
     05 VEND-CODE               PIC X(2).
     05 VENDOR-STATUS           PIC X(2).
     05 VEND-CODE-ERR           PIC X(26).
     05 VEND-STATUS-ERR         PIC X(26).

 01 BUDG-IN                   REDEFINES DATA-BUF.
     05 FILLER                  PIC X(69).
     05 ACCT-NBR                PIC X(20).
     05 BUDG-NBR-ERR            PIC X(26).
     05 BUDG-NY-AMT             PIC X(13).
```

Figure 3

```
       01 FORM-TABLE.
          05 FORM-1.
             10 FILLER                PIC X     VALUE "Z".
             10 FILLER                PIC 9(4) COMP VALUE  1.
             10 FILLER                PIC 9(4) COMP VALUE  1.
             10 FILLER                PIC 9(4) COMP VALUE  14.
          05 FORM-2.
             10 FILLER                PIC X     VALUE "A".
             10 FILLER                PIC 9(4) COMP VALUE  2.
             10 FILLER                PIC 9(4) COMP VALUE  1.
             10 FILLER                PIC 9(4) COMP VALUE  13.
          05 FORM-3.
             10 FILLER                PIC X     VALUE "B".
             10 FILLER                PIC 9(4) COMP VALUE  3.
             10 FILLER                PIC 9(4) COMP VALUE  1.
             10 FILLER                PIC 9(4) COMP VALUE  13.
          05 FORM-4.
             10 FILLER                PIC X     VALUE "C".
             10 FILLER                PIC 9(4) COMP VALUE  4.
             10 FILLER                PIC 9(4) COMP VALUE  1.
             10 FILLER                PIC 9(4) COMP VALUE  11.
          05 FORM-5.
             10 FILLER                PIC X     VALUE "D".
             10 FILLER                PIC 9(4) COMP VALUE  2.
             10 FILLER                PIC 9(4) COMP VALUE  1.
             10 FILLER                PIC 9(4) COMP VALUE  13.
          05 FORM-6.
             10 FILLER                PIC X     VALUE "E".
             10 FILLER                PIC 9(4) COMP VALUE  5.
             10 FILLER                PIC 9(4) COMP VALUE  1.
             10 FILLER                PIC 9(4) COMP VALUE  12.
          05 FORM-7.
             10 FILLER                PIC X     VALUE "Z".
             10 FILLER                PIC 9(4) COMP VALUE  1.
             10 FILLER                PIC 9(4) COMP VALUE  1.
             10 FILLER                PIC 9(4) COMP VALUE  12.
          05 FORM-8.
             10 FILLER                PIC X     VALUE "Z".
             10 FILLER                PIC 9(4) COMP VALUE  1.
             10 FILLER                PIC 9(4) COMP VALUE  1.
             10 FILLER                PIC 9(4) COMP VALUE  14.
          05 FORM-9.
             10 FILLER                PIC X     VALUE "Z".
             10 FILLER                PIC 9(4) COMP VALUE  1.
             10 FILLER                PIC 9(4) COMP VALUE  1.
             10 FILLER                PIC 9(4) COMP VALUE  14.
          05 FORM-10.
             10 FILLER                PIC X     VALUE "Z".
             10 FILLER                PIC 9(4) COMP VALUE  1.
             10 FILLER                PIC 9(4) COMP VALUE  1.
             10 FILLER                PIC 9(4) COMP VALUE  14.

       01 FORM-SPEC-ARRAY           REDEFINES FORM-TABLE.
          05 FORM-SPECS               OCCURS  10 TIMES.
             10 FORM-ID               PIC X.
             10 FORM-NBR              PIC 9(4)  COMP.
             10 FORM-NEXT             PIC 9(4)  COMP.
             10 FORM-HELP             PIC 9(4)  COMP.
```

Figure 4

```
 01 FORM-NAME-TABLE.
     05 FORM-1.
      10 FILLER              PIC X(15) VALUE "BUDMAINT_MENU  ".
      10 FILLER              PIC 9(4)  COMP VALUE  96.
     05 FORM-2.
      10 FILLER              PIC X(15) VALUE "VENDOR_DATA    ".
      10 FILLER              PIC 9(4)  COMP VALUE 357.
     05 FORM-3.
      10 FILLER              PIC X(15) VALUE "BANK_MSTR_DATA ".
      10 FILLER              PIC 9(4)  COMP VALUE 325.
     05 FORM-4.
      10 FILLER              PIC X(15) VALUE "BUDGET_LOAD    ".
      10 FILLER              PIC 9(4)  COMP VALUE 128.
     05 FORM-5.
      10 FILLER              PIC X(15) VALUE "               ".
      10 FILLER              PIC 9(4)  COMP VALUE 96.
     05 FORM-6.
      10 FILLER              PIC X(15) VALUE "               ".
      10 FILLER              PIC 9(4)  COMP VALUE 96.
     05 FORM-7.
      10 FILLER              PIC X(15) VALUE "               ".
      10 FILLER              PIC 9(4)  COMP VALUE  96.
     05 FORM-8.
      10 FILLER              PIC X(15) VALUE "               ".
      10 FILLER              PIC 9(4)  COMP VALUE 96.
     05 FORM-9.
      10 FILLER              PIC X(15) VALUE "               ".
      10 FILLER              PIC 9(4)  COMP VALUE 96.
     05 FORM-10.
      10 FILLER              PIC X(15) VALUE "               ".
      10 FILLER              PIC 9(4)  COMP VALUE 96.
     05 FORM-11.
      10 FILLER              PIC X(15) VALUE "HELP_BANK_DATA ".
      10 FILLER              PIC 9(4)  COMP VALUE 69.
     05 FORM-12.
      10 FILLER              PIC X(15) VALUE "HELP_BUDG_LOAD ".
      10 FILLER              PIC 9(4)  COMP VALUE 69.
     05 FORM-13.
      10 FILLER              PIC X(15) VALUE "HELP_VENDOR    ".
      10 FILLER              PIC 9(4)  COMP VALUE 69.
     05 FORM-14.
      10 FILLER              PIC X(15) VALUE "BUDMAINT_HELP  ".
      10 FILLER              PIC 9(4)  COMP VALUE 69.
     05 FORM-15.
      10 FILLER              PIC X(15) VALUE "HELP_CREDIT_SUP".
      10 FILLER              PIC 9(4)  COMP VALUE 69.

 01 FORM-NAME-ARRAY   REDEFINES FORM-NAME-TABLE.
     05 FORM-NAME-INFO      OCCURS  15 TIMES.
      10 FORM-NAME              PIC X(15).
      10 FORM-DATA-LEN          PIC 9(4)  COMP.
```

Figure 4a

```
102000-EDIT-A.
    MOVE ZERO TO CHECK-RESULT.
    MOVE ZERO TO FIELD-ZERO.
    MOVE  16 TO FIELD-CNT.
    PERFORM 805000-VIEW-EDIT.
    PERFORM 807000-GET-BUFFER.
    MOVE SPACES TO VEND-NBR-ERR, VEND-STATUS-ERR.
    PERFORM 102100-CK-VEND-NBR.
    PERFORM 102200-CK-VEND-CODE.
    PERFORM 102300-CK-VEND-STATUS.
    PERFORM 102400-CK-VEND-1099.

    PERFORM 809000-PUT-BUFFER.
    PERFORM 805000-VIEW-EDIT.
    IF V-NUM-ERRS NOT = ZERO MOVE 1 TO CHECK-RESULT.
    MOVE ZERO TO FIELD-LOC.
    PERFORM 813000-SET-ERROR-FIELDS FIELD-CNT TIMES.

102100-CK-VEND-NBR.
    MOVE VEND-NBR OF VEND-IN TO ARGUMENT.
    PERFORM 831000-GET-VEND-MSTR.
    IF COND-WORD = 17 NEXT SENTENCE
     ELSE MOVE "INVALID! DUPLICATE NUMBER" TO VEND-NBR-ERR
      MOVE  1 TO FIELD-ERR (2)
      MOVE  1 TO CHECK-RESULT.

102200-CK-VEND-CODE.
    IF VEND-CODE OF VEND-IN = "VN" OR = "VM" OR = "DP"
     NEXT SENTENCE
      ELSE MOVE "INVALID VENDOR CODE!" TO VEND-CODE-ERR
       MOVE  1 TO FIELD-ERR (19)
       MOVE  1 TO CHECK-RESULT.

102300-CK-VEND-STATUS.
    IF VENDOR-STATUS OF VEND-IN = "CR" OR = "XX"
     NEXT SENTENCE
      ELSE MOVE "INVALID STATUS CODE!" TO VEND-STATUS-ERR
       MOVE  1 TO FIELD-ERR (20)
       MOVE  1 TO CHECK-RESULT.

102400-CK-VEND-1099.
    IF FLAG-1099 OF VEND-IN = SPACES OR = "Y "
     NEXT SENTENCE
      ELSE
       MOVE  1 TO FIELD-ERR (18)
       MOVE  1 TO CHECK-RESULT.
```

Figure 5

Summary of VIEW Procedures

PROCEDURE	FUNCTION
VCLOSEBATCH	Closes batch file.
VCLOSEFORMF	Closes forms file.
VCLOSETERM	Closes terminal file.
VERRMSG	Returns message associated with error code.
VFIELDEDITS	Edits field data and performs other field processing.
VFINISHFORM	Performs final processing specified for form.
VGETBUFFER	Reads contents of data buffer into user program.
VGETFIELD	Reads field from data buffer into user program.
VGETNEXTFORM	Reads next form into form definition area of memory; window and data buffer are not affected.
VGETtype	Reads field from data buffer to user program, converting data to specified type.
VINITFORM	Sets data buffer to initial values for form.
VOPENBATCH	Opens batch file for processing.
VOPENFORMF	Opens forms file for processing.
VOPENTERM	Opens terminal file for processing.
VPRINTFORM	Prints current form and data on offline list device.
VPUTBUFFER	Writes data from user program to data buffer.
VPUTFIELD	Writes data from user program to field in data buffer.
VPUTtype	Writes data of specified type from user program to data buffer, converting data to ASCII.
VPUTWINDOW	Writes message from user program to window area in memory for later display.
VREADBATCH	Reads record from batch file into data buffer.
VREADFIELDS	Reads input from terminal into data buffer.
VSETERROR	Sets error flag for data field in error; and moves error message to window area.
VSHOWFORM	Updates terminal screen, merging the current form, any data in buffer, and any message in window.
VWRITEBATCH	Writes data from data buffer to batch file.

Figure 6

COBOL VIEW Application Source Program

```
$CONTROL LIST,NOSOURCE,USLINIT,BOUNDS
 IDENTIFICATION DIVISION.
 PROGRAM-ID.
     LAYOUT.
**THIS PROGRAM PROVIDES A LAYOUT FOR USING VIEW FORMS WITH COBOL.
** THE PROGRAM DISPLAYS, EDITS, AND UPDATES MANY FORMS.
*** VERS 0.00     APRIL  9,1980.
 AUTHOR.
     P SOMERS.
 INSTALLATION.
     CAPE DATA INC.
*** (C)COPYRIGHT 1980 CAPE DATA INC. CAPE MAY,NEW JERSEY 08204
 DATE-COMPILED.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. HP-3000.
 OBJECT-COMPUTER. HP-3000.
 INPUT-OUTPUT SECTION.

 DATA DIVISION.

 WORKING-STORAGE SECTION.
 77  FBASE              PIC X(13) VALUE "  BUDGET.PUB;".
 77  PASSWORD           PIC X(8)  VALUE "ABC1234;".
 77  DSET-NAME          PIC X(16) VALUE SPACES.
 77  NO-ITEM            PIC X(2)  VALUE "0;".
 77  ITEM               PIC X(16) VALUE SPACES.
 77  LIST               PIC X(30) VALUE SPACES.
 77  ALL-ITEMS          PIC X(2)  VALUE "@;".
 77  SAME-ITEMS         PIC X(2)  VALUE "*;".
 77  ARGUMENT           PIC X(20) VALUE SPACES.
 77  MODE1              PIC 9(4)  COMP  VALUE 1.
 77  MODE2              PIC 9(4)  COMP  VALUE 2.
 77  MODE3              PIC 9(4)  COMP  VALUE 3.
 77  MODE5              PIC 9(4)  COMP  VALUE 5.
 77  MODE7              PIC 9(4)  COMP  VALUE 7.
 77  MODE-FLAG          PIC X.
 77  LOC-FORM           PIC 9(4)  COMP.
 77  LOC-FORM-NAME      PIC 9(4)  COMP.
 77  LOC-FIND           PIC 9(4)  COMP.
 77  FUTURE-FLAG        PIC 9.
 77  NEW-FLAG           PIC 9.
 77  LAST-RESULT        PIC 9 .
 77  CHECK-RESULT       PIC S9(4)  COMP.
 01  BELL               PIC X   VALUE " ".

 01 STATUS-AREA.
     05 COND-WORD       PIC S9(4)   COMP.
     05 D-L             PIC S9(4)   COMP.
     05 R-N             PIC S9(9)   COMP.
     05 C-L             PIC S9(9)   COMP.
     05 B-A             PIC S9(9)   COMP.
     05 F-A             PIC S9(9)   COMP.

 01 VIEW-COM.
     05 V-STATUS              PIC S9(4)  COMP  VALUE ZERO.
```

```
           05 V-LANGUAGE          PIC S9(4)  COMP  VALUE ZERO.
           05 V-COM-AREA-LEN      PIC S9(4)  COMP  VALUE 60.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 VIEW-MODE           PIC S9(4)  COMP  VALUE ZERO.
           05 LAST-KEY            PIC S9(4)  COMP  VALUE ZERO.
           05 V-NUM-ERRS          PIC S9(4)  COMP  VALUE ZERO.
           05 V-WINDOW-ENH        PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 V-CFNAME            PIC X(15)  VALUE SPACES.
           05 FILLER              PIC X      VALUE SPACES.
           05 V-NFNAME            PIC X(15)  VALUE SPACES.
           05 FILLER              PIC X      VALUE SPACES.
           05 V-REPEAT-OPT        PIC S9(4)  COMP  VALUE ZERO.
           05 V-NF-OPT            PIC S9(4)  COMP  VALUE ZERO.
           05 V-NBR-LINES         PIC S9(4)  COMP  VALUE ZERO.
           05 V-DBUF-LEN          PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 V-DELETE-FLAG       PIC S9(4)  COMP  VALUE ZERO.
           05 V-SHOW-CONTROL      PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 V-NUM-RECS          PIC S9(6)  COMP  VALUE ZERO.
           05 V-REC-NBR           PIC S9(6)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 V-TERM-FILE-NBR     PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.
           05 FILLER              PIC S9(4)  COMP  VALUE ZERO.

       01 V-FILE-NAME       PIC X(36).

       01 ERR-MES-BUF           PIC X(76).
       01 LEN-ERR-BUF           PIC S9(4)  COMP  VALUE 76.
       01 LEN-ERR-MES           PIC S9(4)  COMP  VALUE ZERO.
       01 TERM-FILE             PIC X(8)   VALUE SPACES.
       01 DATA-LEN              PIC S9(4)  COMP .
       01 FIELD-NBR             PIC S9(4)  COMP VALUE ZERO.
       01 ACT-FIELD-LEN         PIC S9(4)  COMP VALUE ZERO.
       01 NEXT-FIELD-NBR        PIC S9(4)  COMP VALUE ZERO.
       01 NO-MESSAGE            PIC S9(4)  COMP VALUE -1.
```

```
 01 NUM-IN                    PIC X(13).
 01 NUM-OUT                   PIC S9(9)V99.
 01 NUM-DISP-13               PIC ---------9.99.


 01 ACCT-MSTR                        COPY ACCTMSTR.

 01 VEND-MSTR                        COPY VENDMSTR.

 01  DATA-BUF.
     05 FILLER          PIC X(69).
     05 DATA-IN         PIC X(443).

 01 MENU-DATA       REDEFINES DATA-BUF.
     05 FILLER                  PIC X(69).
     05 SELECT-IN               PIC X.
     05 SELECT-ERR            PIC X(26).

 01 VEND-IN       REDEFINES DATA-BUF.
     05 FILLER                  PIC X(69).
     05 VEND-NBR                PIC X(6).
     05 VEND-NBR-ERR            PIC X(26).
     05 VEND-NAME               PIC X(30).
     05 S-ADDRESS               PIC X(30).
     05 S-ADDRESS2              PIC X(30).
     05 S-CITY                  PIC X(20).
     05 S-STATE                 PIC X(2).
     05 S-ZIP                   PIC X(10).
     05 P-ADDRESS               PIC X(30).
     05 P-ADDRESS2              PIC X(30).
     05 P-CITY                  PIC X(20).
     05 P-STATE                 PIC X(2).
     05 P-ZIP                   PIC X(10).
     05 PHONE-NBR.
      10 PHONE-AC               PIC X(3).
      10 PHONE-EX               PIC X(3).
      10 PHONE-NO               PIC X(4).
     05 FLAG-1099               PIC X(2).
     05 VEND-CODE               PIC X(2).
     05 VENDOR-STATUS           PIC X(2).
     05 VEND-CODE-ERR           PIC X(26).
     05 VEND-STATUS-ERR         PIC X(26).

 01 BUDG-IN                   REDEFINES DATA-BUF.
     05 FILLER                  PIC X(69).
     05 ACCT-NBR                PIC X(20).
     05 BUDG-NBR-ERR            PIC X(26).
     05 BUDG-NY-AMT             PIC X(13).

*****************************************************
***          FORM TABLE LAYOUT:
***                SELECTION CHARACTER
***                FORM NUMBER
***                NEXT FORM NUMBER
***                HELP FORM NUMBER
***
*****************************************************
```

```

      ****************************************************
      ***             FORM NAME TABLE LAYOUT:
      ***                   FORM NAME (VIEW FORM NAME)
      ***                   LENGTH OF DATA FIELDS
      ***
      ****************************************************


       01 FORM-TABLE.
           05 FORM-1.
             10 FILLER                      PIC X      VALUE "Z".
             10 FILLER                      PIC 9(4) COMP VALUE  1.
             10 FILLER                      PIC 9(4) COMP VALUE  1.
             10 FILLER                      PIC 9(4) COMP VALUE  14.
           05 FORM-2.
             10 FILLER                      PIC X      VALUE "A".
             10 FILLER                      PIC 9(4) COMP VALUE  2.
             10 FILLER                      PIC 9(4) COMP VALUE  1.
             10 FILLER                      PIC 9(4) COMP VALUE  13.
           05 FORM-3.
             10 FILLER                      PIC X      VALUE "B".
             10 FILLER                      PIC 9(4) COMP VALUE  3.
             10 FILLER                      PIC 9(4) COMP VALUE  1.
             10 FILLER                      PIC 9(4) COMP VALUE  13.
           05 FORM-4.
             10 FILLER                      PIC X      VALUE "C".
             10 FILLER                      PIC 9(4) COMP VALUE  4.
             10 FILLER                      PIC 9(4) COMP VALUE  1.
             10 FILLER                      PIC 9(4) COMP VALUE  11.
           05 FORM-5.
             10 FILLER                      PIC X      VALUE "D".
             10 FILLER                      PIC 9(4) COMP VALUE  2.
             10 FILLER                      PIC 9(4) COMP VALUE  1.
             10 FILLER                      PIC 9(4) COMP VALUE  13.
           05 FORM-6.
             10 FILLER                      PIC X      VALUE "E".
             10 FILLER                      PIC 9(4) COMP VALUE  5.
             10 FILLER                      PIC 9(4) COMP VALUE  1.
             10 FILLER                      PIC 9(4) COMP VALUE  12.
           05 FORM-7.
             10 FILLER                      PIC X      VALUE "Z".
             10 FILLER                      PIC 9(4) COMP VALUE  1.
             10 FILLER                      PIC 9(4) COMP VALUE  1.
             10 FILLER                      PIC 9(4) COMP VALUE  12.
           05 FORM-8.
             10 FILLER                      PIC X      VALUE "Z".
             10 FILLER                      PIC 9(4) COMP VALUE  1.
             10 FILLER                      PIC 9(4) COMP VALUE  1.
             10 FILLER                      PIC 9(4) COMP VALUE  14.
           05 FORM-9.
             10 FILLER                      PIC X      VALUE "Z".
             10 FILLER                      PIC 9(4) COMP VALUE  1.
             10 FILLER                      PIC 9(4) COMP VALUE  1.
             10 FILLER                      PIC 9(4) COMP VALUE  14.
           05 FORM-10.
             10 FILLER                      PIC X      VALUE "Z".
```

```
               10 FILLER                PIC 9(4) COMP VALUE  1.
               10 FILLER                PIC 9(4) COMP VALUE  1.
               10 FILLER                PIC 9(4) COMP VALUE  14.

       01 FORM-SPEC-ARRAY              REDEFINES FORM-TABLE.
             05 FORM-SPECS              OCCURS  10 TIMES.
               10 FORM-ID               PIC X.
               10 FORM-NBR              PIC 9(4)  COMP.
               10 FORM-NEXT             PIC 9(4)  COMP.
               10 FORM-HELP             PIC 9(4)  COMP.

       01 FORM-NAME-TABLE.
             05 FORM-1.
               10 FILLER              PIC X(15) VALUE "BUDMAINT_MENU  ".
               10 FILLER              PIC 9(4)  COMP VALUE  96.
             05 FORM-2.
               10 FILLER              PIC X(15) VALUE "VENDOR_DATA    ".
               10 FILLER              PIC 9(4)  COMP VALUE 357.
             05 FORM-3.
               10 FILLER              PIC X(15) VALUE "BANK_MSTR_DATA ".
               10 FILLER              PIC 9(4)  COMP VALUE 325.
             05 FORM-4.
               10 FILLER              PIC X(15) VALUE "BUDGET_LOAD    ".
               10 FILLER              PIC 9(4)  COMP VALUE 128.
             05 FORM-5.
               10 FILLER              PIC X(15) VALUE "               ".
               10 FILLER              PIC 9(4)  COMP VALUE 96.
             05 FORM-6.
               10 FILLER              PIC X(15) VALUE "               ".
               10 FILLER              PIC 9(4)  COMP VALUE 96.
             05 FORM-7.
               10 FILLER              PIC X(15) VALUE "               ".
               10 FILLER              PIC 9(4)  COMP VALUE  96.
             05 FORM-8.
               10 FILLER              PIC X(15) VALUE "               ".
               10 FILLER              PIC 9(4)  COMP VALUE 96.
             05 FORM-9.
               10 FILLER              PIC X(15) VALUE "               ".
               10 FILLER              PIC 9(4)  COMP VALUE 96.
             05 FORM-10.
               10 FILLER              PIC X(15) VALUE "               ".
               10 FILLER              PIC 9(4)  COMP VALUE 96.
             05 FORM-11.
               10 FILLER              PIC X(15) VALUE "HELP_BANK_DATA ".
               10 FILLER              PIC 9(4)  COMP VALUE 69.
             05 FORM-12.
               10 FILLER              PIC X(15) VALUE "HELP_BUDG_LOAD ".
               10 FILLER              PIC 9(4)  COMP VALUE 69.
             05 FORM-13.
               10 FILLER              PIC X(15) VALUE "HELP_VENDOR    ".
               10 FILLER              PIC 9(4)  COMP VALUE 69.
             05 FORM-14.
               10 FILLER              PIC X(15) VALUE "BUDMAINT_HELP  ".
               10 FILLER              PIC 9(4)  COMP VALUE 69.
             05 FORM-15.
               10 FILLER              PIC X(15) VALUE "HELP_CREDIT_SUP".
               10 FILLER              PIC 9(4)  COMP VALUE 69.
```

```

 01 FORM-NAME-ARRAY   REDEFINES FORM-NAME-TABLE.
     05 FORM-NAME-INFO       OCCURS  15 TIMES.
      10 FORM-NAME              PIC X(15).
      10 FORM-DATA-LEN          PIC 9(4)  COMP.







 01 FORMAT-CNTL                  COPY FORMTCTL.

 01 FORMAT-13.
     05 FILLER               PIC S9(4)  COMP VALUE 1.
     05 FILLER               PIC S9(4)  COMP VALUE 13.
     05 FILLER               PIC S9(4)  COMP VALUE 0.
     05 FILLER               PIC X      VALUE " ".
     05 FILLER               PIC X      VALUE SPACES.
     05 D13-FORMAT.
      10 FILLER              PIC S9(4)  COMP VALUE 11.
      10 FILLER              PIC S9(4)  COMP VALUE ZERO.
      10 FILLER              PIC X      VALUE SPACES.
      10 FILLER              PIC X      VALUE "N".
      10 FILLER              PIC X      VALUE "1".
      10 FILLER              PIC X      VALUE "2".


 01 ERROR-ARRAY.
     05 FIELD-ERR          OCCURS 56 TIMES   PIC 9.
 01 FIELD-ZERO      REDEFINES ERROR-ARRAY   PIC X(56).

 01 FIELD-CNT              PIC 99  COMP.
 01 FIELD-LOC              PIC 99  COMP.

*****************************************************************
$PAGE


 PROCEDURE DIVISION .
 000000-MAIN-PART SECTION 01.
 000000-PROGRAM-LOGIC.
     PERFORM 900000-OPEN-PROGRAM.
     PERFORM 100000-READ-LOOP UNTIL LAST-KEY = 8.
     PERFORM 970000-CLOSE-PROGRAM.
     STOP RUN.

 100000-READ-LOOP.
     PERFORM 801000-READ-TERM.
     IF LAST-KEY = 0
        NEXT SENTENCE
      ELSE IF LAST-KEY = 1
        PERFORM 881000-KEY-1
      ELSE IF LAST-KEY = 2
        PERFORM 882000-KEY-2
      ELSE IF LAST-KEY = 3
```

```
               PERFORM 883000-KEY-3
             ELSE IF LAST-KEY = 4
               PERFORM 884000-KEY-4
             ELSE IF LAST-KEY = 5
               PERFORM 885000-KEY-5
             ELSE IF LAST-KEY = 6
               PERFORM 886000-KEY-6
             ELSE IF LAST-KEY = 7
               PERFORM 887000-KEY-7
             ELSE
               PERFORM 888000-KEY-8.

            IF CHECK-RESULT NOT = 0  OR LAST-KEY = 5
               NEXT SENTENCE
              ELSE IF LOC-FORM =  1
               PERFORM 101000-EDIT-MENU
             ELSE IF LOC-FORM =  2
               PERFORM 102000-EDIT-A
             ELSE IF LOC-FORM =  3
               PERFORM 103000-EDIT-B
             ELSE IF LOC-FORM =  4
               PERFORM 104000-EDIT-C
             ELSE IF LOC-FORM =  5
               PERFORM 105000-EDIT-D
             ELSE IF LOC-FORM =  6
               PERFORM 106000-EDIT-E
             ELSE
               MOVE  4 TO CHECK-RESULT.

            IF CHECK-RESULT NOT = ZERO
               NEXT SENTENCE
             ELSE IF LOC-FORM =  1
               PERFORM 150000-VALID-MENU
             ELSE IF LOC-FORM =  2
               PERFORM 151000-VALID-A
             ELSE IF LOC-FORM =  3
               PERFORM 152000-VALID-B
             ELSE IF LOC-FORM =  4
               PERFORM 153000-VALID-C
             ELSE IF LOC-FORM =  6
               PERFORM 155000-VALID-E
             ELSE IF LAST-KEY NOT = 5
               PERFORM 889000-ASK-CONFIRM
             ELSE IF LOC-FORM =  5
               PERFORM 154000-VALID-D
             ELSE
               MOVE 4 TO CHECK-RESULT.


            IF CHECK-RESULT = 3 PERFORM 851000-REFRESH-TERM.
            IF CHECK-RESULT = 4 MOVE 3 TO V-SHOW-CONTROL
             MOVE "Z" TO MODE-FLAG.
            IF CHECK-RESULT = 0 PERFORM 811000-FORM-INITIALIZE.
            IF CHECK-RESULT = 2 OR = 4 OR = 6 PERFORM 852000-NEXT-FORM.
            IF CHECK-RESULT NOT = 9
               PERFORM 804000-SHOW-FORM
               PERFORM 803000-CLEAR-WINDOW.
```

```
           MOVE ZERO TO CHECK-RESULT, V-SHOW-CONTROL.


       101000-EDIT-MENU.
           MOVE ZERO TO CHECK-RESULT.
           PERFORM 805000-VIEW-EDIT.
           PERFORM 807000-GET-BUFFER.
           MOVE SPACES TO SELECT-ERR.
           IF SELECT-IN  = "X" MOVE 8 TO LAST-KEY
              MOVE 9 TO CHECK-RESULT
            ELSE
              PERFORM 101100-EDIT-MENU-DATA.

       101100-EDIT-MENU-DATA.
           PERFORM 101110-MENU-SCAN VARYING LOC-FIND FROM 1  BY  1
            UNTIL LOC-FIND >  15
             OR SELECT-IN = FORM-ID (LOC-FIND).
           IF LOC-FIND >  15 PERFORM 101120-MENU-ERROR
            ELSE PERFORM 803000-CLEAR-WINDOW.

       101110-MENU-SCAN.
           EXIT.

       101120-MENU-ERROR.
           MOVE
            "PLEASE SELECT ONE OF THE ABOVE LETTERS AND RE-ENTER"
            TO ERR-MES-BUF.
           MOVE 51  TO LEN-ERR-BUF.
           MOVE  2  TO FIELD-NBR.
           PERFORM 812000-SET-ERROR.
           ADD  1 TO V-NUM-ERRS.
           MOVE 1 TO CHECK-RESULT.


      ****************************************************************
      *                     MODE SELECTIONS                          *
      *    A = ADD VENDOR MASTER                  N =                *
      *    B = ADD BANK MASTER                    O =                *
      *    C = ADD NEXT YR.BUDGET                 P =                *
      *    D = EDIT VENDOR MASTER                 Q =                *
      *    E = LOAD BUDGET                        R =                *
      *    F =                                    S =                *
      *    G =                                    T =                *
      *    H =                                    U =                *
      *    I =                                    V =                *
      *    J =                                    W =                *
      *    K =                                    X =  EXIT  PROGRAM *
      *    L =                                    Y =                *
      *    M =                                    Z = MAIN MENU      *
      *                                                              *
      *                                                              *
      ****************************************************************


      ***********************************************************
      ***                CHECK RESULT  CODES                  ***
      ***      0 = NO ERRORS                     5 =          ***
```

```
***      1 = ERRORS, RE-READ           6 = NEXT FORM          ***
***      2 = BRING UP NEW FORM         7 =                    ***
***      3 = REFRESH TERMINAL          8 =                    ***
***      4 = RETURN TO MAIN MENU       9 = EXIT PROGRAM       ***
***                                                           ***
*****************************************************************


**********************************************************************
***          FUNCTION  KEY CODES                                  ***
***    F1 =  SKIP                     F5  = CONFIRM ENTRY         ***
***    F2 = CLEAR (INITIALIZE)        F6  = NEXT FORM             ***
***    F3 = HELP                      F7  = MAIN MENU             ***
***    F4 = REFRESH SCREEN            F8  = EXIT                  ***
***                                                               ***
**********************************************************************

 102000-EDIT-A.
     MOVE ZERO TO CHECK-RESULT.
     MOVE ZERO TO FIELD-ZERO.
     MOVE  16 TO FIELD-CNT.
     PERFORM 805000-VIEW-EDIT.
     PERFORM 807000-GET-BUFFER.
     MOVE SPACES TO VEND-NBR-ERR, VEND-STATUS-ERR.
     PERFORM 102100-CK-VEND-NBR.
     PERFORM 102200-CK-VEND-CODE.
     PERFORM 102300-CK-VEND-STATUS.
     PERFORM 102400-CK-VEND-1099.

     PERFORM 809000-PUT-BUFFER.
     PERFORM 805000-VIEW-EDIT.
     IF V-NUM-ERRS NOT = ZERO MOVE 1 TO CHECK-RESULT.
     MOVE ZERO TO FIELD-LOC.
     PERFORM 813000-SET-ERROR-FIELDS FIELD-CNT TIMES.

 102100-CK-VEND-NBR.
     MOVE VEND-NBR OF VEND-IN TO ARGUMENT.
     PERFORM 831000-GET-VEND-MSTR.
     IF COND-WORD = 17 NEXT SENTENCE
      ELSE MOVE "INVALID! DUPLICATE NUMBER" TO VEND-NBR-ERR
       MOVE  1 TO FIELD-ERR (2)
       MOVE  1 TO CHECK-RESULT.

 102200-CK-VEND-CODE.
     IF VEND-CODE OF VEND-IN = "VN" OR = "VM" OR = "DP"
      NEXT SENTENCE
       ELSE MOVE "INVALID VENDOR CODE!" TO VEND-CODE-ERR
        MOVE  1 TO FIELD-ERR (19)
        MOVE  1 TO CHECK-RESULT.

 102300-CK-VEND-STATUS.
     IF VENDOR-STATUS OF VEND-IN = "CR" OR = "XX"
      NEXT SENTENCE
       ELSE MOVE "INVALID STATUS CODE!" TO VEND-STATUS-ERR
        MOVE  1 TO FIELD-ERR (20)
        MOVE  1 TO CHECK-RESULT.
 102400-CK-VEND-1099.
     IF FLAG-1099 OF VEND-IN = SPACES OR = "Y "
      NEXT SENTENCE
       ELSE
        MOVE  1 TO FIELD-ERR (18)
        MOVE  1 TO CHECK-RESULT.




 103000-EDIT-B.
     MOVE ZERO TO CHECK-RESULT.
     MOVE ZERO TO FIELD-ZERO.
     MOVE  3 TO FIELD-CNT.
     PERFORM 805000-VIEW-EDIT.
     PERFORM 807000-GET-BUFFER.
     PERFORM 103100-CK-VEND-NBR.
     PERFORM 102200-CK-VEND-CODE.
     PERFORM 102300-CK-VEND-STATUS.
     PERFORM 102400-CK-VEND-1099.
     PERFORM 809000-PUT-BUFFER.
     PERFORM 805000-VIEW-EDIT.
     IF V-NUM-ERRS  NOT = ZERO
        MOVE  1 TO CHECK-RESULT.
     MOVE ZERO TO FIELD-LOC.
     PERFORM 813000-SET-ERROR-FIELDS FIELD-CNT TIMES.

 103100-CK-VEND-NBR.
     IF NEW-FLAG = ZERO NEXT SENTENCE
      ELSE IF VEND-NBR OF VEND-IN = VEND-NBR OF VEND-MSTR
        MOVE  1 TO NEW-FLAG
      ELSE MOVE ZERO TO NEW-FLAG.
     MOVE VEND-NBR OF VEND-IN TO ARGUMENT.
     PERFORM 831000-GET-VEND-MSTR.
     IF COND-WORD = ZERO AND NEW-FLAG = 1
        NEXT SENTENCE
      ELSE IF COND-WORD = ZERO
        PERFORM 103110-SETUP-VENDOR
      ELSE
        MOVE "NON-EXISTENT VENDOR NUMBER!" TO VEND-NBR-ERR
        MOVE  1 TO FIELD-ERR (2)
        MOVE  1 TO CHECK-RESULT.

 103110-SETUP-VENDOR.
     MOVE CORR VEND-MSTR TO VEND-IN.
     MOVE  1 TO NEW-FLAG, CHECK-RESULT.

 104000-EDIT-C.
     MOVE ZERO TO CHECK-RESULT.
     MOVE ZERO TO FIELD-ZERO.
     MOVE  3 TO FIELD-CNT.
     PERFORM 805000-VIEW-EDIT.
     PERFORM 807000-GET-BUFFER.
*    MOVE SPACES TO ERROR DISPLAY FIELDS
*    PERFORM EDIT ROUTINES
     PERFORM 809000-PUT-BUFFER.
     PERFORM 805000-VIEW-EDIT.
```

```
          IF V-NUM-ERRS  NOT = ZERO
             MOVE  1 TO CHECK-RESULT.
          MOVE ZERO TO FIELD-LOC.
          PERFORM 813000-SET-ERROR-FIELDS FIELD-CNT TIMES.

      105000-EDIT-D.
          MOVE ZERO TO CHECK-RESULT.
          MOVE ZERO TO FIELD-ZERO.
          MOVE  3 TO FIELD-CNT.
          PERFORM 805000-VIEW-EDIT.
          PERFORM 807000-GET-BUFFER.
   *      MOVE SPACES TO ERROR DISPLAY FIELDS
   *      PERFORM EDIT ROUTINES
          PERFORM 809000-PUT-BUFFER.
          PERFORM 805000-VIEW-EDIT.
          IF V-NUM-ERRS  NOT = ZERO
             MOVE  1 TO CHECK-RESULT.
          MOVE ZERO TO FIELD-LOC.
          PERFORM 813000-SET-ERROR-FIELDS FIELD-CNT TIMES.

      106000-EDIT-E.
          MOVE ZERO TO CHECK-RESULT.
          MOVE ZERO TO FIELD-ZERO.
          MOVE  3 TO FIELD-CNT.
          PERFORM 805000-VIEW-EDIT.
          PERFORM 807000-GET-BUFFER.
   *      MOVE SPACES TO ERROR DISPLAY FIELDS
   *      PERFORM EDIT ROUTINES
          PERFORM 809000-PUT-BUFFER.
          PERFORM 805000-VIEW-EDIT.
          IF V-NUM-ERRS  NOT = ZERO
             MOVE  1 TO CHECK-RESULT.
          MOVE ZERO TO FIELD-LOC.
          PERFORM 813000-SET-ERROR-FIELDS FIELD-CNT TIMES.


      150000-VALID-RECORD SECTION 03.
      150000-VALID-MENU.
          MOVE 2 TO CHECK-RESULT.
          MOVE LOC-FIND TO LOC-FORM.
          MOVE ZERO TO NEW-FLAG.

      151000-VALID-A.

      152000-VALID-B.

      153000-VALID-C.

      154000-VALID-D.

      155000-VALID-E.




      800000-UTILITIES SECTION 02.

```

```
 801000-READ-TERM.
     CALL "VREADFIELDS" USING VIEW-COM.
     IF V-STATUS NOT = 0 PERFORM 992000-VIEW-ERROR.

 802000-PUT-WINDOW.
     CALL "VPUTWINDOW" USING VIEW-COM ERR-MES-BUF LEN-ERR-BUF.

 803000-CLEAR-WINDOW.
     MOVE SPACES TO ERR-MES-BUF.
     PERFORM 802000-PUT-WINDOW.


 804000-SHOW-FORM.
     CALL "VSHOWFORM" USING VIEW-COM.
     IF V-STATUS NOT = 0 PERFORM 992000-VIEW-ERROR.


 805000-VIEW-EDIT.
     CALL "VFIELDEDITS" USING VIEW-COM.

 806000-FINISH-FORM.
     CALL "VFINISHFORM" USING VIEW-COM.

 807000-GET-BUFFER.
     CALL "VGETBUFFER" USING VIEW-COM DATA-BUF V-DBUF-LEN.
     IF V-STATUS NOT = 0 PERFORM 992000-VIEW-ERROR.

 809000-PUT-BUFFER.
     CALL "VPUTBUFFER" USING VIEW-COM DATA-BUF V-DBUF-LEN.
     IF V-STATUS NOT = 0 PERFORM 992000-VIEW-ERROR.

 810000-GET-FORM-FILE.
     MOVE 0  TO V-REPEAT-OPT.
     MOVE 0  TO V-NF-OPT.
     CALL "VGETNEXTFORM" USING VIEW-COM.
     IF V-STATUS NOT = 0 PERFORM 992000-VIEW-ERROR.

 811000-FORM-INITIALIZE.
     MOVE 65  TO V-WINDOW-ENH.
     CALL "VINITFORM" USING VIEW-COM.
*    ADD INDIVIDUAL FORMS INITIALIZATION HERE AS REQ.
     IF V-STATUS NOT = 0 PERFORM 992000-VIEW-ERROR.
     PERFORM 803000-CLEAR-WINDOW.
     MOVE  1 TO LAST-RESULT.

 812000-SET-ERROR.
     CALL "VSETERROR" USING VIEW-COM FIELD-NBR ERR-MES-BUF
      LEN-ERR-BUF.
     IF V-STATUS NOT = 0 PERFORM 992000-VIEW-ERROR.

 813000-SET-ERROR-FIELDS.
     ADD 1  TO FIELD-LOC.
     IF FIELD-ERR (FIELD-LOC) = 1
        CALL "VSETERROR" USING VIEW-COM FIELD-LOC ERR-MES-BUF
        NO-MESSAGE
          IF V-STATUS NOT = ZERO
            PERFORM 992000-VIEW-ERROR
```

```
               ELSE
                 NEXT SENTENCE
            ELSE
              NEXT SENTENCE.

       814000-CONFIRM-READ.
           CALL "IMMVREADFIELDS" USING VIEW-COM.
           IF V-STATUS NOT = ZERO  PERFORM 992000-VIEW-ERROR.


       820000-SPACE-NUMBER.
           MOVE FORMAT-13 TO FORMAT-CNTL.
           CALL "CAPE'ENTRY" USING FORMAT-CNTL NUM-IN NUM-OUT.
           IF CFIELD-ERR (1) = ZERO AND CENTRY-ERR = ZERO
            MOVE NUM-OUT TO NUM-DISP-13
             ELSE MOVE 1 TO CHECK-RESULT.

       830000-GET-ACCT-MSTR.
           MOVE "ACCOUNT-MSTR;" TO DSET-NAME.
           CALL "DBGET" USING FBASE DSET-NAME MODE7 STATUS-AREA
            ALL-ITEMS ACCT-MSTR ARGUMENT.
           IF COND-WORD NOT = ZERO AND NOT = 17
              PERFORM 991000-STATUS-CK
            ELSE
              NEXT SENTENCE.

       831000-GET-VEND-MSTR.
           MOVE "VENDOR-MSTR;" TO DSET-NAME.
           CALL "DBGET" USING FBASE DSET-NAME MODE7 STATUS-AREA
            ALL-ITEMS VEND-MSTR ARGUMENT.
           IF COND-WORD = ZERO OR = 17
              NEXT SENTENCE
            ELSE
              PERFORM 991000-STATUS-CK.

       841000-DB-LOCK.
           CALL "DBLOCK" USING FBASE DSET-NAME MODE3 STATUS-AREA.
           IF COND-WORD NOT = ZERO PERFORM 991000-STATUS-CK.

       842000-DB-UNLOCK.
           CALL "DBUNLOCK" USING FBASE DSET-NAME MODE1 STATUS-AREA.
           IF COND-WORD NOT = ZERO PERFORM 991000-STATUS-CK.



       851000-REFRESH-TERM.
           PERFORM 980000-CLOSE-TERM.
           PERFORM 902000-OPEN-TERM.
           MOVE 3 TO V-SHOW-CONTROL.
           PERFORM 811000-FORM-INITIALIZE.

       852000-NEXT-FORM.
           IF CHECK-RESULT =  4 MOVE  1  TO LOC-FORM.
           MOVE FORM-NBR (LOC-FORM) TO LOC-FORM-NAME.
           MOVE FORM-NAME (LOC-FORM-NAME) TO V-NFNAME.
           PERFORM 810000-GET-FORM-FILE.
           PERFORM 811000-FORM-INITIALIZE.
```

```
      853000-EDIT-ERROR.
           CALL "VERRMSG" USING VIEW-COM ERR-MES-BUF LEN-ERR-BUF
            LEN-ERR-MES.
           IF V-STATUS NOT = 0 PERFORM 992000-VIEW-ERROR.
           CALL "VPUTWINDOW" USING VIEW-COM ERR-MES-BUF LEN-ERR-BUF.

      854000-PUT-TITLE.
     *     MOVE 1 TO FIELD-NBR.
     *     CALL "VPUTFIELD" USING VIEW-COM FIELD-NBR TITLE-BUF
     *      TITLE-LEN ACT-FIELD-LEN NEXT-FIELD-NBR.
     *     IF V-STATUS NOT = 0 PERFORM 992000-VIEW-ERROR.



      860000-START-HELP.
           MOVE FORM-HELP (LOC-FORM) TO LOC-FORM-NAME.
           MOVE FORM-NAME (LOC-FORM-NAME)  TO V-NFNAME.

      861000-HELP-DISPLAY.
           PERFORM 810000-GET-FORM-FILE.
           PERFORM 854000-PUT-TITLE.
           PERFORM 804000-SHOW-FORM.
           CALL "VREADFIELDS" USING VIEW-COM.
           IF V-STATUS NOT = 0 PERFORM 992000-VIEW-ERROR.
           IF LAST-KEY = 8
              MOVE  9 TO CHECK-RESULT
            ELSE IF LAST-KEY = 7
              MOVE  4 TO CHECK-RESULT
            ELSE IF LAST-KEY = 4
              MOVE  3 TO CHECK-RESULT
            ELSE
              MOVE 2 TO CHECK-RESULT.


      865000-INITIAL-VENDOR.
     *     MOVE VEND-NBR OF VEND-MSTR TO VEND-NBR OF VEND-IN.
     *     MOVE VEND-NAME OF VEND-MSTR TO VEND-NAME OF VEND-IN.
     *     PERFORM 809000-PUT-BUFFER.
     *

      870000-UPDATE-ACCT-MSTR.
           CALL "DBUPDATE" USING FBASE DSET-NAME MODE1
            STATUS-AREA ALL-ITEMS ACCT-MSTR.
           IF COND-WORD = ZERO NEXT SENTENCE
            ELSE PERFORM 991000-STATUS-CK.



      881000-KEY-1.
           MOVE 1 TO CHECK-RESULT.
           MOVE "INVALID KEY SELECTED, IGNORED" TO ERR-MES-BUF.
           MOVE 29 TO LEN-ERR-BUF.
           PERFORM 802000-PUT-WINDOW.

      882000-KEY-2.
           MOVE 1 TO CHECK-RESULT.
           PERFORM 811000-FORM-INITIALIZE.

       883000-KEY-3.
           PERFORM 860000-START-HELP THRU 861000-HELP-DISPLAY.

       884000-KEY-4.
           MOVE 3 TO CHECK-RESULT.

       885000-KEY-5.
           IF LAST-RESULT = ZERO
              PERFORM 814000-CONFIRM-READ
              MOVE  5 TO LAST-KEY
              MOVE ZERO TO CHECK-RESULT
            ELSE
              PERFORM 890000-INVALID-CONFIRM.

       886000-KEY-6.
           MOVE 6 TO CHECK-RESULT.
           MOVE FORM-NEXT (LOC-FORM ) TO LOC-FORM.


       887000-KEY-7.
           MOVE "Z" TO MODE-FLAG.
           MOVE 4 TO CHECK-RESULT.

       888000-KEY-8.
           MOVE 9 TO CHECK-RESULT.

       889000-ASK-CONFIRM.
           MOVE
            "VALID RECORD, PUSH CONFIRM KEY (F5) TO POST AS SHOWN"
             TO ERR-MES-BUF.
           MOVE 52  TO LEN-ERR-BUF.
           PERFORM 802000-PUT-WINDOW.
           MOVE  1 TO CHECK-RESULT.
           MOVE ZERO TO LAST-RESULT.

       890000-INVALID-CONFIRM.
           MOVE  1 TO CHECK-RESULT.
           MOVE
            "INVALID USE OF CONFIRM KEY!  CONFIRM NOT REQUESTED!"
            TO ERR-MES-BUF.
           MOVE  52 TO LEN-ERR-BUF.
           PERFORM 802000-PUT-WINDOW.

      ***********************************************************

       900000-START-STOP SECTION 51.
       900000-OPEN-PROGRAM.
           DISPLAY "VIEW/COBOL LAYOUT PROGRAM  VERS. 0.01".
           PERFORM 901000-OPEN-DATA-BASE.
           PERFORM 902000-OPEN-TERM.
           PERFORM 903000-OPEN-VFORM.
           PERFORM 904000-START-MENU.

       901000-OPEN-DATA-BASE.
           CALL "DBOPEN" USING FBASE PASSWORD MODE1 STATUS-AREA.
           IF COND-WORD = ZERO
              NEXT SENTENCE
            ELSE
              PERFORM 991000-STATUS-CK
              STOP RUN.

       902000-OPEN-TERM.
           MOVE ZERO TO V-STATUS, V-LANGUAGE,
            VIEW-MODE, LAST-KEY, V-NUM-ERRS, V-REPEAT-OPT
            V-NF-OPT.
           CALL "VOPENTERM" USING VIEW-COM TERM-FILE.
           IF V-STATUS = ZERO
              NEXT SENTENCE
            ELSE
              PERFORM 992000-VIEW-ERROR
              CALL "VCLOSEFORMF" USING VIEW-COM
              DISPLAY ERR-MES-BUF  " STOPPING RUN! "
              PERFORM 990000-STOP-PAR.

       903000-OPEN-VFORM.
           MOVE "BUDGFORM.BUDGET.PROGLIB" TO V-FILE-NAME.
           CALL "VOPENFORMF" USING VIEW-COM V-FILE-NAME.
           IF V-STATUS = ZERO
              NEXT SENTENCE
            ELSE
              PERFORM 992000-VIEW-ERROR
              CALL "VCLOSETERM" USING VIEW-COM
              DISPLAY ERR-MES-BUF  " STOPPING RUN! "
              PERFORM 990000-STOP-PAR.

       904000-START-MENU.
           MOVE  1  TO LOC-FORM.
           PERFORM 852000-NEXT-FORM.
           PERFORM 804000-SHOW-FORM.
           MOVE ZERO TO CHECK-RESULT.


       970000-CLOSE-PROGRAM.
           PERFORM 980000-CLOSE-TERM.
           PERFORM 981000-CLOSE-VFORM.
           PERFORM 990000-STOP-PAR.

       980000-CLOSE-TERM.
           CALL "VCLOSETERM" USING VIEW-COM.
           IF V-STATUS NOT = 0 PERFORM 992000-VIEW-ERROR.

       981000-CLOSE-VFORM.
           CALL "VCLOSEFORMF" USING VIEW-COM.
           IF V-STATUS NOT = ZERO PERFORM 992000-VIEW-ERROR.


       990000-STOP-PAR.
           CALL "DBCLOSE" USING FBASE DSET-NAME MODE1 STATUS-AREA.
           STOP RUN.


       991000-STATUS-CK.
```

```
           PERFORM 980000-CLOSE-TERM.
           PERFORM 981000-CLOSE-VFORM.
           CALL "DBEXPLAIN" USING STATUS-AREA.
           PERFORM 990000-STOP-PAR.

       992000-VIEW-ERROR.

           CALL "VERRMSG" USING VIEW-COM ERR-MES-BUF LEN-ERR-BUF
            LEN-ERR-MES.
           DISPLAY BELL "VIEW ERROR!!!".
           DISPLAY ERR-MES-BUF.
           DISPLAY BELL "PROGRAM TERMINATED DUE TO ABOVE ERROR!!!".
           PERFORM 990000-STOP-PAR.

```

Process Sensing and Control

Nancy Kolitz
Hewlett-Packard Company
Cupertino, California

I. INTRODUCTION

Various MPE intrinsics on the HP3000 allow a user to create processes, to obtain information about them, and to control them. This paper will describe the process sensing and control capabilities available to a user, through illustrations and examples. The paper will also introduce a new intrinsic, PROCINFO, currently being developed by the MPE lab.

II. WHAT IS A PROCESS?

All user programs run as processes under MPE. A process is the unique execution of a program by a particular user at a particular time, and is the entity within MPE which can accomplish work. A process is also the mechanism which allows system resources to be shared and a user's code to be executed. Each process consists of a private data stack and code segments, shared by all processes executing the same program.

As the system is brought up, the Progenitor (PROGEN) is the first process created by MPE. One of the various system processes that Progen creates is the User Controller Process (UCOP), which creates a User Main Process (UMAIN) as a session or job logs on. Then when a user (or job) runs a program, a User Son of Main (USONM) process is created. If other processes are subsequently created from this program, User processes are established. (See Figure 1.)

```
              PROGEN
                |
               UCOP
              / | \
             /  |  \
            /   |   \
           /    |    \
      UMAIN   UMAIN   UMAIN
                |
              USONM
              / | \
             /  |  \
         USER  USER  USER .....
```

Figure 1

A process will be in one of two states once it has been created: Wait or Active. If it is in a wait state, it is waiting for some event (I/O, RIN acquistion, etc.) to occur before it will run again. If it is in an active state, the process is running or ready to run.

A standard MPE user has no control over his processes. The operating system creates, controls, and kills the processes for the user. However, if the user's program has Process Handling (PH) capability, it can, to some degree, manage its own processes. In fact, it can even control processes in its family tree.

III. PROCESS CREATION

In MPE, there are two intrinsics that a user with PH capability can use to create a process: CREATE and CREATEPROCESS.

The intrinsic CREATE will load a program into virtual memory, create a new process, initialize the stack's data segment, schedule the process to run, and return the process identification (PIN) number to the process requesting the creation. Once the process is established, it will have to be activated by the creating process. The command syntax is:

```
           BA       BA       I   IV   LV     IV     IV
CREATE(progname,entryname,pin,param,flags,stacksize,dlsize,
         IV        LV       IV      O-V
       maxdata,priorityclass,rank).
```

The last parameter RANK (in the CREATE intrinsic) is not used by the intrinsic and is only there for compatibility with previous versions of MPE.

CREATEPROCESS is the other intrinsic that can be used for creating processes. Its format is:

```
                    I     I       BA          IA         LA        O-V
CREATEPROCESS(error,pin,progname,itemnums,items).
```

The parameter ITEMNUMS indicates the options to be applied in creating the new process, and the parameter ITEMS provides the necessary information to be used for each option specified in ITEMNUMS.

With CREATEPROCESS, a son may be activated immediately upon creation or may be activated as a process is with CREATE (via the ACTIVATE intrinsic). A user may also specify an entry point into a program, define $STDIN and $STDLIST to be any file other than the defaults (the defaults are the creating father's $STDIN and $STDLIST), control stack size, and control the process' priority queue. Some of these can also be done with CREATE.

The example that follows illustrates the intrinsic CREATEPROCESS. It will create a process, indicate that the father should be awakened upon completion of the son, and then activate the new process.

****

```
BEGIN
          <<CREATEPROCESS example>>

INTEGER ERROR, PIN;
BYTE ARRAY EXAMPLE(0:7) := "EXAMPLE ";
INTEGER ARRAY OPTNUMS(0:10);
LOGICAL ARRAY OPTIONS(0:10);

intrinsic CREATEPROCESS,TERMINATE;

          <<set up options>>
OPTNUMS(0) := 3;           OPTIONS(0) := 1;
OPTNUMS(1) := 10;          OPTIONS(1) := 3;
OPTNUMS(2) := 0;    <<terminator>>

CREATEPROCESS(ERROR,PIN,EXAMPLE,OPTNUMS,OPTIONS);

if <> then TERMINATE;
```

****

When calling MPE intrinsics, a good programming practice is to check the condition code returned, and the error parameter, if one is used. In the case of CREATEPROCESS, if the condition code is less than zero the process was created, but some event occurred to cause the operating system to give a warning to the creator. If the condition code is greater than zero, an error has occurred and the process was not created. If the error occurred because of a file system problem (error number returned is 6), a user can use the intrinsic FCHECK with a parameter of zero to obtain more information as to why the process creation failed.

IV. SENSING PROCESSES

Each process in MPE has a large amount of information about it that can be useful, providing a process can access it. There are various intrinsics that will return this information once a process has been created. However, a program must have PH capability to use these intrinsics.

A user may determine the PIN number of the process that created it via the intrinsic FATHER. Its syntax is:

```
I
pin := FATHER.
```

Once again, a programmer should check the condition code that was returned. In this case, it will tell what type of process the father is. Through specific codes, it will specify whether the father is a system process, a user main process, or a user process.

To obtain the PIN number of any of his son processes, a program may use the intrinsic GETPROCID. The command is:

```
 I                 IV
pin := GETPROCID(numson).
```

The parameter NUMSON is a integer value that specifies which son a father wants to know the PIN number. For example, if a father has created three sons and wants to know the PIN number of the second son, he will supply GETPROCID with a parameter of two.

The WHO intrinsic provides the access mode and attributes of the user running a program. The file access capabilities (save file (SF), ability to access nonsharable devices (ND), etc.), user attributes (OP, SM, etc), and user capabilities (PH, DS, etc) can be obtained. Also information about the user, his logon group name and account name, his home group, and the logical device of his input file may be returned. The command syntax for WHO is:

```
      L        D          D        BA       BA      BA      BA      L       O-V
WHO(mode,capability,lattr,usern,groupn,acctn,homen,termn).
```

The intrinsic GETORIGIN will return, to a reactivated process, the origin of its activation. The value returned will specify if the PIN was activated from a suspended state by a father, a son, or another source (interrupt or timer). GETORIGIN looks like:

```
  I
source := GETORIGIN.
```

Other information about a son or father may be obtained from the intrinsic GETPROCINFO. Its format is:

```
 D                       IV
statinfo := GETPROCINFO(pin).
```

A double word is passed back giving the process' priority number and priority queue, its activity state (active or waiting), its suspension condition and source of next activation, and the origin of its last activation. The process number, passed as a parameter, specifies which process you want information about. If PIN=0, then information is returned for the father; otherwise, the information is for a son process.

A new intrinsic currently under development in the MPE lab is called PROCINFO. This intrinsic returns general information about processes that is currently unavailable, unless you have privileged mode capability. It should simplify some of the uses of process related intrinsics because a large amount of information may be retrieved in one call to PROCINFO. Its command syntax is:

```
           I       I      IV      I      BA         O-V
PROCINFO(error1,error2,pin[,itemnum1,item1]
                          [,itemnum2,item2]
                          [,itemnum3,item3]
                          [,itemnum4,item4]
                          [,itemnum5,item5]
                          [,itemnum6,item6]).
```

This intrinsic is formatted similar to FFILEINFO in order to maintain ease of use and extensibility. It can return to a program the process number of the process itself, its father, all its sons, and all its descendants. It can also supply information about the number of descendants and generations in a family tree, the name of a program that a specified process is running, the process' state, and the process' priority number.

The first error word is used to return the type of error incurred when executing the intrinsic. The second error word returns the index of the offending item number. The program name is returned in a byte array that is a minimum of twenty eight bytes long. It is in the format of <filename.group.account>.

The following example will help to illustrate the use of the PROCINFO intrinsic:

```
     ***********************************************

BEGIN  <<procinfo example>>

  INTEGER ERROR1, ERROR2, PIN;

  BYTE ARRAY ITEMVAL1 (0:1),
             ITEMVAL2 (0:1),
             ITEMVAL3 (0:1),
             ITEMVAL4 (0:1),
             ITEMVAL5 (0:1);

  INTEGER ITEMNUM1,ITEMNUM2,ITEMNUM3,ITEMNUM4,ITEMNUM5;


  INTRINSIC PROCINFO;

  PIN := 0;              <<seek information about ourselves>>
  ITEMNUM1 := 1;      <<request our pin #>>
  ITEMNUM2 := 3;      <<how many sons we have>>
  ITEMNUM3 := 4;      <<how many descendants we have>>
  ITEMNUM4 := 2;      <<pin number of our father>>
  ITEMNUM5 := 5;      <<how many generations we have>>

  PROCINFO (ERROR1, ERROR2, PIN, ITEMNUM1, ITEMVAL1,
                                 ITEMNUM2, ITEMVAL2,
                                 ITEMNUM3, ITEMVAL3,
                                 ITEMNUM4, ITEMVAL4,
                                 ITEMNUM5, ITEMVAL5);

  IF <> THEN GO PROCERROR;


  .  .  .  .

  PROCERROR:
    <<print message and error number>>
    RETURN;

END.  <<procinfo example>>


     ***********************************************
```

If the previous program was executed by pin 45 in the process tree of figure 2, the following information would be returned:

```
item number  information
    1            45
    3            2
    4            5
    2            12
    5            3
```

```
                 Pin 12
                  /   \
                 /     \
                /       \
           Pin 23      Pin 45
                        /   \
                       /     \
                      /       \
                 Pin 22      Pin 34
                  /  |  \
                 /   |   \
                /    |    \
               /     |     \
              /      |      \
         Pin 38  Pin 21  Pin 30
```

Figure 2

****

V. CONTROLLING PROCESSES

Once a program has created a process, it can control its activity. As mentioned before, it can activate its sons via the intrinsic ACTIVATE. However, only a father can activate a newly created process. ACTIVATE is called with the following parameters:

```
                IV    IV        O-V
      ACTIVATE (pin, susp).
```

The process' pin number is required, but the susp parameter is not. If susp is provided and not equal to zero, then the calling process will be suspended and the specified process will be activated. Otherwise, the father process continues to run and the activated process becomes ready to run. The activated process will execute when the dispatcher selects it as the highest priority process to launch.

A process may also suspend itself. Via the intrinsic SUSPEND, a process may place itself in a wait state and state its expected origin of activation. The intrinsic calling sequence is:

```
              LV    IV          O-V
      SUSPEND (susp,rin).
```

The RIN parameter is the Resource Identification Number that will be locked for the process until it suspends again. The RIN allows a process to have exclusive access to a particular resource at a particular time. This is one way to synchronize processes and their resources running under the same job.

One other process control intrinsic is GETPRIORITY. When a process is created, it is given the same priority as its father. This intrinsic allows a program to change its own process' priority or that of a son. The process cannot, however, request a priority outside of its allowable priority class. GETPRIORITY is called as follows:

```
                 IV         LV          IV        O-V
      GETPRIORITY(pin,priorityclass,rank).
```

The priorityclass parameter is a 16-bit word that contains two ASCII characters. Depending on the priority queue desired, the parameter is "AS," "BS," "CS," "DS," or "ES." (If a user has privileged mode, he can supply an absolute number for the priority parameter instead of the ASCII characters. It is done by supplying the parameter "xA where "x" is an integer value and "A" is the ASCII character A.) The rank parameter, once again, is not used except for compatibility with old versions of MPE.

The last two intrinsics to be discussed are used for process termination. When a process is terminated, it must return all the system resources that it is holding, stop its sons from running and start their termination sequence, and then request that its father take away its stack. The two intrinsics used for termination are:

```
     IV
KILL(pin)       and         TERMINATE.
```

The parameter in the KILL intrinsic is the pin number of the process' son that it wants deleted. TERMINATE can only be used for the calling process.

The following is another example using these various intrinsics. This example illustrates the CREATE, ACTIVATE, GETPRIORITY and TERMINATE intrinsics:

```
*************************************************

BEGIN

ARRAY NAME(0:15) := "EXAMPLE.PUB.SYS ";
BYTE ARRAY BNAME (*) = NAME;
INTEGER PIN;

INTRINSIC CREATE,ACTIVATE,TERMINATE,GETPRIORITY;

CREATE(BNAME,,PIN,,1);        <<create the new process. reactivate >>
                              <<the father when this one finishes.>>

IF <> THEN TERMINATE;         <<kill process because of error in>>
                              <<creation sequence              >>

ACTIVATE(PIN,2);              <<activate process and then reactivate>>
                              <<calling process by the son          >>

IF <> THEN TERMINATE;         <<process not activated due to error>>

GETPRIORITY(PIN,"DS");        <<change priority of son process>>

IF <> THEN TERMINATE;         <<new priority not granted>>

END.

*************************************************
```

VI. SUMMARY

This paper has summarized various intrinsics that can be used to create new processes, obtain information about them, control them, and then terminate them. A new intrinsic, PROCINFO, was also introduced which can provide the user with more information about processes without requiring privileged mode capability. MPE is a process oriented operating system, and with a better knowledge and understanding of how processes operate, a user can enhance his applications and their performance on the HP3000.

Putting the HP3000 to Work For Programmers

Thomas L. Fraser
Forest Computer Incorporated
East Lansing, MI

I. THE OPPORTUNITY

The demand for software is exploding as businesses and other organizations which use computers strive to be more productive, control costs, and improve the quality of management information. The acceleration of this demand is forecasted to continue throughout the early 1980s.

Software is produced for the most part by people, skilled people. These "programmers" are a limited resource. If the increasing demand is to be met, either the size of this resource must be increased or the productivity of the resource must be improved.

Looking at the issue from the viewpoint of an individual DP shop, increasing the size of the resource means hiring people. Skilled people are expensive, and costs are going up. Especially expensive are programmers, due to the already existing shortage. This shortage also makes it difficult to find quality people. So increasing the size of the resource is not always easy and is very costly.

Another trend that is evident is the decreasing cost of computer hardware. This contributes to the increasing demand for software, and thus is part of the problem. However, it can be made part of the solution by putting computers to work for the programmers.

This is the opportunity. Use the computer to increase the productivity of programmers. Provide software tools which allow the people to work efficiently and quickly. The expensive and scarce programmer should not have to wait for or adapt to the increasingly inexpensive computer. In a word, the computer needs to be made more friendly toward the programmer.

II. THE HYPOTHESIS

In the specific environment of the HP3000, programming is usually done online. A majority of the programs are written in COBOL with FORTRAN also popular. There are several types of tools which can be introduced to this environment. Report generators, high level file systems, COBOL generators, forms generators, and very high level languages such as RAPID/3000 can all help. However, in most shops programmers still spend a large amount of time at a terminal working with source code. This therefore is the first place to look when considering how to get the HP3000 working for the programmer.

By far the most prevalent software tool used by programmers is the HP editor. Compared to the primeval batch methods of source input and maintenance, EDIT/3000 is vastly superior. Because the editor is interactive, changes can be viewed as they are made within the context of the rest of the program. Also the editor provides many features such as searches and global changes previously unavailable. And best of all there is no problem keeping card decks in sequence.

However, the new features and capabilities come with a price, that of increased demand on the system resources. The programmers are competing with each other, as well as with production users, for precious disk accesses and CPU time. An obvious result of any delay in system response is lower productivity. This applies to all users of the system, including programmers.

EDIT/3000 is not without weaknesses. It is a line-by-line editor. This is a logical carryover from the days of cards. (Remember, the VDT was originally intended as a keypunch replacement.) All I/O is organized around the line as a standard unit. I/O from the terminal interrupts the hardware once for each character because of lack of block-mode handling. Moreover, the software must get involved each time "RETURN" is hit; this is a minimum of once per line with the exception of the "CHANGE" command, and with many commands can be several times per line. Disk I/O is blocked, but the binary search used to locate the card-image formatted records i very expensive in terms of disk accesses. This line orientation has obvious negative performance implications. Moreover, it means that the programmer must work with a line at a time. Despite the ability to display 20 lines on a single CRT screen, only one line at best can be entered or changed per transmission except for the noted exception.

Believing that the overall demand on resources might be reduced, and system performance improved, there still remain other areas to be investigated when seeking to improve upon the editor. For example, the "TEXT" and "KEEP" commands are very slow due to the fact they are actually file copying commands.

One of the nice features of EDIT/3000, that of being able to see the changes in context, is mitigated against by two major factors. The first is screen clutter. Unless one repeatedly does "LIST" commands, the screen becomes full of old source lines and already executed commands as well as current source lines. The second is the inability to access everything on the screen.

Performing some operations, even on a single source line, require several commands to be transmitted. This makes more effort by the programmer necessary, and slows the coding process. This, together with the other factors, are seriously impairing the speed of software development and system performance.

Thus the hypothesis, that a full screen block-mode editor, written for maximum features with minimum demand on machine resources, would dramatically improve programmer productivity. Improved response time for other users could also be anticipated.

III. THE METHOD

To test the hypothesis a full screen, block-mode editor was designed and written. The result of this effort, called "CHICKEN" by its architects, was a COBOL and SPL program which can be used to edit source code, documentation, stream-files, and other text. No operating system modifications are required, and the program runs in ordinary session-mode.

Block-mode transmissions dramatically reduce the overhead of terminal I/O. This is especially true when the line is driven from the Asynchronous Data Communications Controller (ADCC). The number of transmissions is also reduced making life easier for the programmer. The terminal has a microprocessor; block-mode enables taking advantage of this to reduce load on the HP3000. If one has paid for a "smart terminal," it behooves one to use it. By the way, CHICKEN can automatically switch the terminal between block-mode and character as needed. Implicit is that the VDT being used is an HP compatible terminal with block-mode capability.

Full screen access is another way of putting the terminal to work. With the new editor, all twenty-four lines of the screen are used. One line is for entering commands, one line for error messages, and the other twenty-two are used to display source lines. The programmer can change, delete, or insert lines of code any place on the screen by using just the terminal capabilities. Only after completing an entire screen, is the source transmitted to the HP3000. At that time CHICKEN will determine which lines should be deleted, changed, or added to the file. There is no need to use commands to tell it what is a change, delete, etc.

The disk organization of source files also effects significant advantages. Standard MPE files are used, but CHICKEN has its own access techniques. The old card-image format is replaced by a compressed format which is designed to maximize performance while using less disk space. Because of the file organization and access methods, CHICKEN can retrieve any single line of source code in one disk access, and any twenty-two consecutive lines in an average of 1.4 seeks with a maximum of two required. This single technique has great performance implications.

Ease of use is always an important design consideration and CHICKEN is easy to use. The command set uses language similar to EDIT/3000 to make it easy to quickly get acquainted. Any command can be issued at any time. Moreover, it is seldom necessary to issue multiple commands to accomplish a single task. Recall also, that the software frequently will figure out what you want done without having to be specifically told. Probably the biggest factor in ease of use, though, is the full screen access. A simple list of commands is below.

CHICKEN has other features which contribute to improved productivity:

- Screens are automatically formatted for COBOL, FORTRAN or SPL source if desired.
- The programmer has access to most MPE commands from the editor.
- Compiles can be submitted without leaving the editor.
- Special passwords are put on source files.
- An optional log of changes provides a means of recovery and a means of "backing out" modifications. This also can be used to provide an audit trail.

Several commands are listed below to show general syntax and to compare their operation with the similar commands available in EDIT/3000. In general, the commands follow a standard format as shown here:

```
CMD <starting-line <ending-line>> required-params
                 <optional-params>
```

```
CMD <starting-line <ending-line>> required-params
                 <optional-params>
```

Most command key-words are the same as found in EDIT/3000, and all can be invoked by entering only the first letter. For instance, "LIST 120.5" can be entered as "L 120.5".

CHICKEN attempts to give the user as much flexibility in entering a command as possible, so as to accommodate differing user styles acquired through exposure to various other editors. Thus the following commands would all have the same effect if entered:

```
DELETE 20/30
D (20.00:30.00)
DEL 20 30
DELETE 20,30
```

The goal here is to make the editor easy to learn by not requiring strict adherence to particular syntax rules, and easy to remember by keeping command formats simple and regular.

Following are some representative commands:

```
TEXT edit-file < NEW < mpe-source-file > >
```

This command opens and grants access to an edit-file.

If another edit-file is currently open and being worked on, it is automatically closed. If the NEW option is entered, a new edit-file is created. The "mpe-source-file" refers to an EDIT/3000 source file which can be copied to the CHICKEN edit-file. This command executes very quickly because there is no copy operation from a source file to a work file as in EDIT/3000, except when an MPE source file is copied in, which happens only rarely.

KEEP < A < B > > mpe-source-file < PURGE >

This command makes a copy of the currently accessed edit-file to an EDIT/3000 formatted source file. Normally all lines will be copied. If line A is specified, all lines from line A through the end of the edit-file will be copied. If line B is specified, the copy will only include the lines from line A through line B. If the PURGE option is entered, the edit-file is closed and purged from the system after a successful copy operation.

This command is used infrequently, usually for backup purposes. Since compiles can be implemented directly from within the editor on the existing edit-files, there just isn't much need to KEEP files. If one edit-file is TEXTed in and modified, a second TEXT automatically closes the first edit-file with changes intact. The improvement in response time to access edit-files can be dramatic even on only a moderately loaded system.

LIST < { A / LAST } >

A simple LIST command without parameters will display the first 22 lines of text in the edit-file. Subsequent transmission will display the next 22 lines, in effect paging through the text. If line A is specified, then line A and the next 21 lines of text following line A will be displayed. Again, paging applies after entering the command once. If "LAST" is specified, then the last line of text and 21 blank lines are displayed.

This is where some of the power and flexibility of a full screen block-mode editor can be seen. The user can now be free to move the cursor anywhere on the screen, modifying, inserting, and deleting lines. Changes can be reviewed in context of the surrounding text. Even line numbers can be changed simply by typing over the old ones displayed. All of this goes on without bothering the host computer. Of course, this frees up the HP3000 for other tasks at hand.

FIND < A < B > > *textl* < ALL >

This command performs a search for the next occurrence of textl and displays the line containing the textl along with the following 21 lines of text. If line A is specified, the search will begin at line A and continue until a match is found or the end of the file is reached. If line B is specified, the search will only encompass lines A through B. The asterisks surrounding textl represent delimiters, which can be any non-alphanumeric characters including a space.

Examples:

FIND MEN	<spaces as delimiters>
F/ALL MEN/	<slashes as delimiters . . . space is part of the search string>

The ALL option will cause the editor to attempt to find all occurrences of textl and display all corresponding lines. If 22 occurrences are found before the search line limit, the lines containing occurrences of textl are displayed along with a message stating that the search is not finished. The user can modify any of the lines on the screen. To resume the search, the user only needs to enter "F", and the editor picks up the search where it left off. The user can even begin a search and then use other commands such as LIST or CHANGE, add and delete lines, etc., and will still be able to resume a search.

RENUMBER < A < B > > < BY N >

Renumbers the edit-file. If no parameters are entered, all text lines are renumbered. If line A is specified, the numbering will begin at line A and continue through the end of the file. If line B is specified, the renumbering will only be done on lines A through B. The BY N option allows the user to override the default line number increments used by CHICKEN in a renumber operation.

The renumbering is done to the file in place, rather than through a copy procedure. This significantly speeds up the operation in comparison to, say, EDIT/3000's GATHER command.

Other commands found in EDIT/3000 as well as many other line editors, are either unnecessary or have their utility reduced with a full screen editor. The ADD command in EDIT/3000 is a good example. With CHICKEN lines are inserted right on the screen between other lines of text, and transmitted back to the editor. The line number does not even have to be included. The editor identifies the surrounding text and calculates a line number for the new line. To add text at the end of a file, the user enters L LAST. CHICKEN displays the last line of text followed by 21 numbered blank text lines. Along the same vein, line deletes can be handled on the screen simply by placing the letter "D" before a displayed line of text. The DELETE command itself is only needed for global deletes, as in D 100/200.

The above are just a sampling of the full screen editor's command list. As was mentioned previously, command keywords have, for the most part, been kept the same to facilitate learning to use CHICKEN. Thus the user will find such familiar key-words as GATHER, JOIN, HOLD, CHANGE, EXIT, etc., along with a few new commands, such as ZIP which initiates a compile for an edit-file without having to either exit the editor or KEEP the source code.

THE RESULT
(A Personal Digression)

The most notable difference upon the installation of

CHICKEN was not programmer productivity, it was programmer euphoria. After using it for even a short time, one gets hooked. In our shop we have a mixture of block-mode and character-mode terminals used for program development. To say that the character-mode terminals are collecting dust would be an exaggeration, but we have noticed people arriving quite early in the morning to stake claim to a "CHICKEN" terminal.

The productivity, response time, and performance improvements are also accomplished. As of this writing (December 1981), quantitative data are not available. Anyone desiring more information of this nature, or having any further interest in learning more about CHICKEN can write to:

Tom Fraser
Forest Computer
P.O. Box 1010
East Lansing, Michigan 48823

or call (517) 332-7777.

Techniques for Testing On-Line Interactive Programs

Kim D. Leeper
Wick Hill Associates Ltd.
Kirkland, Washington

ABSTRACT

This paper will describe various strategies for testing on-line interactive programs. These strategies include acceptance/functional testing, regression testing and contention,testing. The paper will also discuss the mechanics of testing including testing by human intervention and various forms of automated testing. This information will allow you to create a viable test plan for software quality assurance in your shop.

INTRODUCTION

Program Testing. Those two words undoubtedly conjure up thoughts of long boring hours sittig in front of a terminal typing in all kinds of data looking at error messages produced by the program. This paper will present alternatives to this type of program testing. It will also describe a prototype test plan or quality assurance cycle which may provide the reader with ideas for implementing his/her own test plan for his/her own shop.

We must make sure we are all talking the same language so some definitions are in order at this point.

What is Testing?

Software testing may be thought of as a series of data items which when presented to the program under test (PUT) cause the software in question to react in a prescribed or expected fashion within its intended environment. The purpose of testing is to expose the existence of mistakes in the program or to show the absence of any such bugs. If the software does not act in the expected way then one has found a bug or mistake in the program.

Vocabulary

SCRIPT — a list of inputs or data items given to the PUT for testing purposes.

DATA CONTEX OF BUG — the collection of inputs required to cause the PUT to fail or return results which are not expected.

TYPES OF TESTING

Acceptance/Functional Testing

This type of testing is used to demonstrate that the various functions of a given software package actually works as described in its documentation. This is not exhaustive testing as it only examines one or two transactions per function. This is the typical type of testing the vast majority of users perform now.

Regression Testing

This type of testing can be used to test all the various logical paths within a given software system. Regression testing tries all the data extremes per function that the program could be expected to respond to. This type of testing is rarely performed because it is resource, that is to say hardware and personnel, intensive.

Contention Testing

This type of testing is used to determine if the database or file locking strategies that are used in your application programs actually work. Two programs are executed at the same time, one performs a transaction which locks a given item in the database. The second program attempts to access this same data that is secured by the lock via another transaction type different than the one used in the first terminal. The designer in this instance is interested in the message of action of the software to this challenge. This type of testing becomes particularly relevant when the installation has many programmers implementing many systems dealing with the same database.

THE TEST PLAN OR QUALITY ASSURANCE CYCLE

The keystone of any successful testing program is to have a viable test plan. This plan should describe all the phases a software development project goes through and then ties all the phases together in one comprehensive flow of data and actions. The plan should extensively use feedback loops so that when problems are discovered there are clear paths for the problem rectification process to follow. One possible quality assurance cycle that can be proposed may be seen in Figure 1.

The diagram indicates that the test script should be generated along with the design of the software. Many times in the design process the designer realizes some weakness in the design and will want to specify a special test in the scriptfile. S/he is encouraged to do so. Many companies that use this methodology specify programs by a test script and V/3000 screens.

Examining this diagram more closely one can see that the flow of debugging actions is closely tied to the design/maintenance of the original test script. The reason for this is to force the implementors to keep track of the bugs they discover and place them in the test script. This script should then be run against the application program whenever a new fix or correction has been applied to the original program. This script will constantly force the program to re-execute all the previous transactions which caused bugs to occur in the past, to assure the program maintenance team that no additional mistakes have been introduced by fixing the last bug.

In this version of the QA cycle the users are always in a mode of testing the delivered software. Eventually the users will find a bug which will start the whole cyclic process over again. If they don't find a bug, don't think it is not for trying. The users have eight hours per day per person to find bugs. It does not take very long before they have more execution time on the application software than the designer/implementator has. This is the time when more bugs can and will be found which will start the cycle once again.

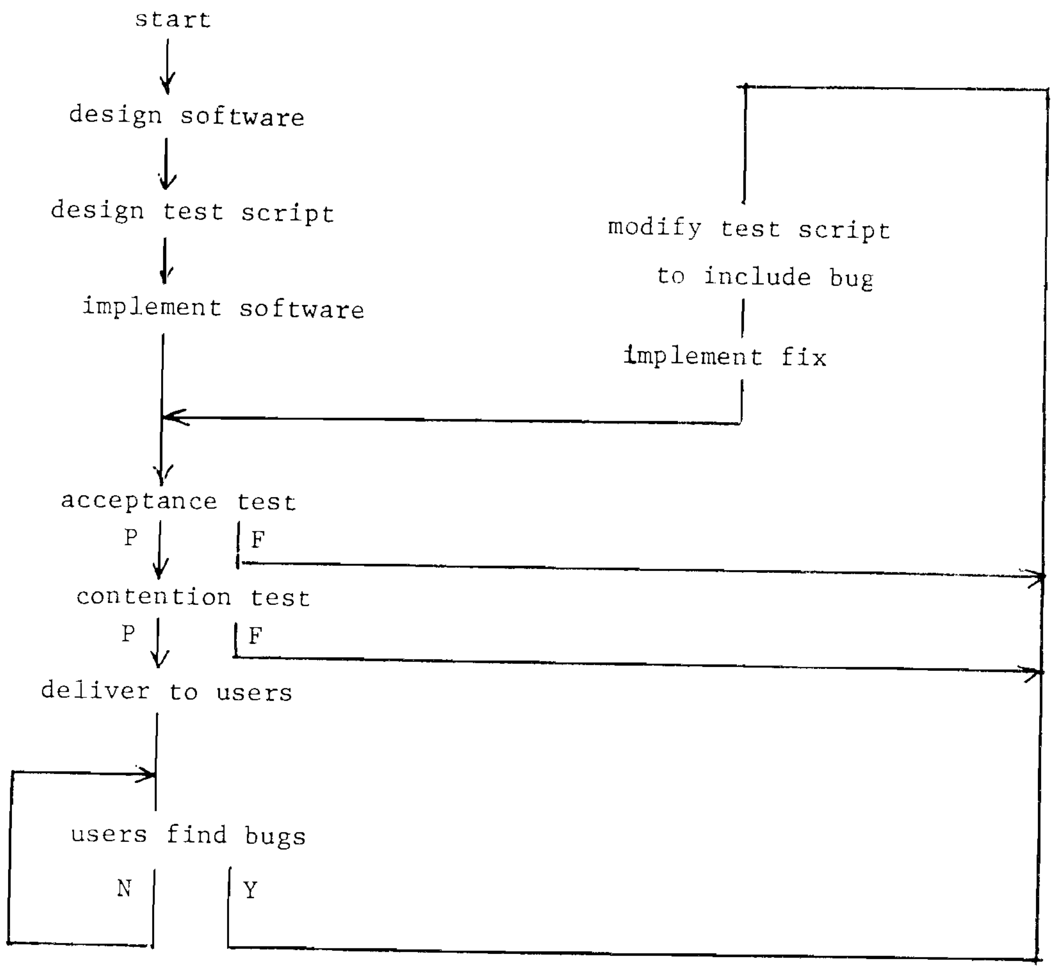


Figure 1
Quality Assurance Software Cycle

THE MECHANICS OF TESTING

Obviously, the type of testing that is currently being used is human intervention testing. This is where a programmer of analyst sits in front of a terminal and simulates a user by following a handwritten script. This approach to testing is less than desirable for a number of reasons, among those being:

1. input data error due to arrogance/boredom in applications tester;
2. non-repeatability of exact timing due to human tester;
3. the tester might not record everything happening off the screen;
4. an expensive employee is being utilized for testing purposes when s/he could be designing/implementing more applications

A possible solution to the dilemma outlined above is to mechanially examine the software by exhaustively testing all the paths in the program by computer. Using completely random data types as input you could automate the testing process. However, as there is only so much time available during a 24 hour day it might take all day to exhaustively test a very small application program. This technique is machine bound in terms of both creating the random data and testing all the paths in the application code.

A saner approach would be to combine the above two techniques into a testing procedure that utilizes a human being's capacity for creative thought and a machine's capacity for highly efficient repetition. This technique would rest in the programmers designing the scripts used for automated testing at the same time as they design the application itself. Once the test script is produced then the machine itself tests out the application program under the watchful eye of a human. In fact the script can be used as a specification for implementing the system. As Yourdon has written, "What we are interested in is the minimum volume of test data that will adequately exercise our program."[1]

It is now possible, using VTEST/3000, to automate this testing procedure and achieve a real manner of quality control. VTEST/3000 includes full V/3000 testing capability. The compiled code runs as though it were in a live situation with VTEST/3000 providing full documentation of all errors occurring on the screen of the terminal.

In order to use VTEST effectively one must appreciate the diagram in Figure 2. There are two types of tests that VTEST can perform, block mode testing for those programs that use V/3000 and non-block mode testing for those not using V.

The first type of testing that will be discussed is non-block mode application testing. In this case VTEST looks like a non-block mode glass TTY terminal. The script file contains the actual commands and data that a user would normally type into the screen of a real terminal, everything between and including HELLO and BYE. This script file is built and maintained by the standard HP EDITOR. The script file is input to VTEST. VTEST transmits this file a line at a time to the application and VTEST prints out a report of the terminal screen before the return key was depressed and after along with the number of seconds that the response took to come back to VTEST.

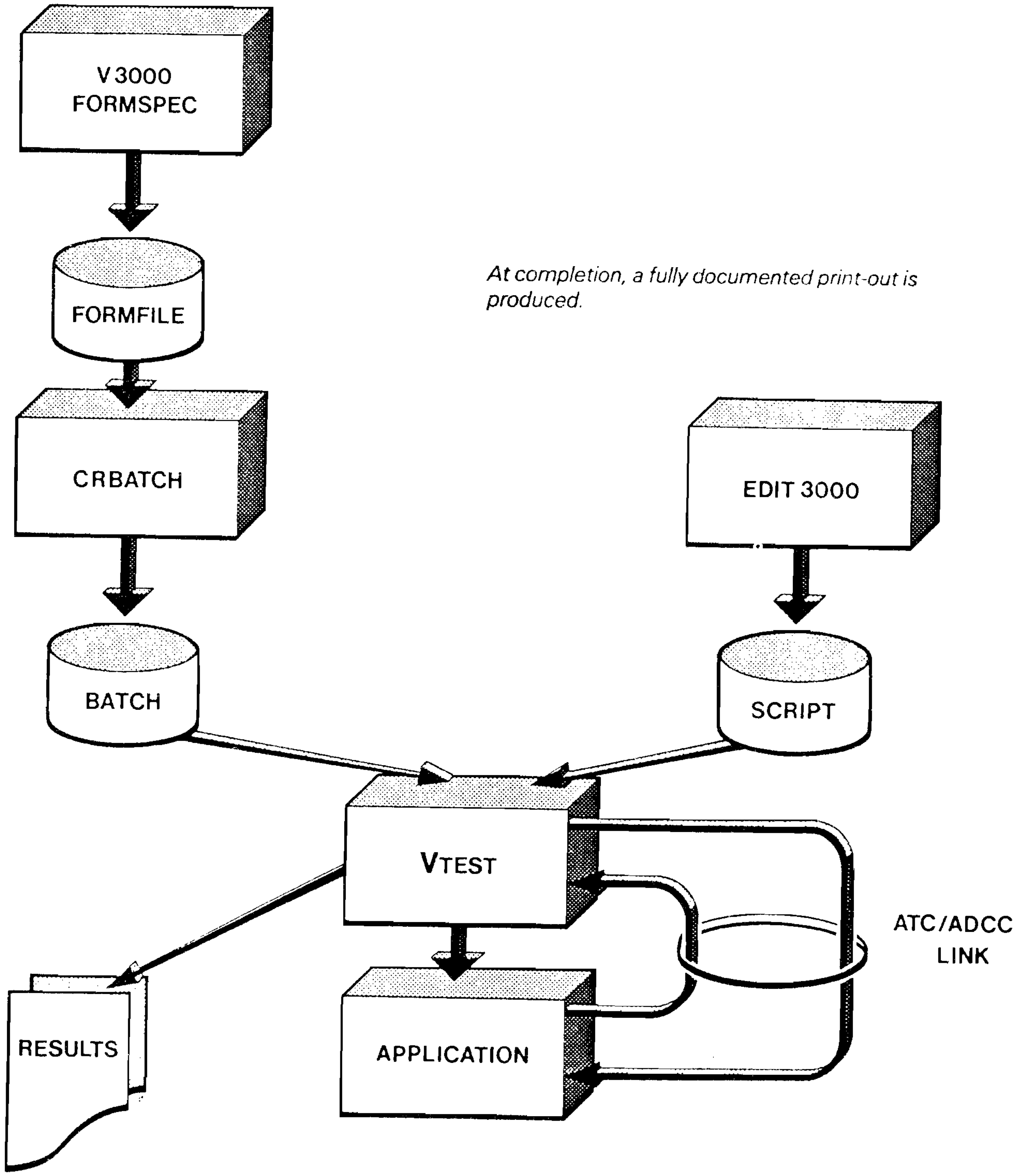
The second type of testing that will be discussed is block mode application testing. In this case VTEST looks like a HP2645 block mode terminal. The script file is the same as above with an important extension. The script file now can tell VTEST when it must transmit data to a V screen. The data for a V screen must come from a different type of file. This file is called the BATCH file. This BATCH file is created and maintained by another program called CRBATCH. CRBATCH allows the user to specify the formfile name and the form to be displayed. Data is then entered and CRBATCH reads the screen and puts the data into a BATCH file. CRBATCH allows the user to insert screens, to delete screens and modify the data in screens already in the BATCH file. It is a general purpose maintenance program or editor for BATCH files. Whenever the application program under test wants some block mode data the next record is read from the BATCH file. VTEST then transmits this record complete with all the special characters that V requires to the application. VTEST prints out a report for every transaction before the ENTER key was depressed and after the next screen was received along with the number of seconds that the response took to come back to VTEST.

One can see quite easily that VTEST fits right into a well designed quality assurance cycle.

REFERENCES

[1]Edward Yourdon, "Techniques of Program Structure and Design," Prentice-Hall, 1975.

[2]Software Research Associates, "Testing Techniques Newsletter," (415) 957-1441.



At completion, a fully documented print-out is produced.

Figure 2

A Universal Approach an an Alternative to Conventional Programming

Bill McAfee and Craig Winters
Futura Systems
Austin

Some two years ago we set about to find a shortcut to programming, a way to simplify and speedup the actual coding, to eliminate all or nearly all of the housekeeping, and to improve the reliability and maintain-ability of our work. We wanted to be able to deal with any problem in terms of the logical operations to solve it, rather than with a sequence of detailed programming statements.

We identified approximately 100 routines to handle input, validation, conversion, formatting, and other functions not provided in the System Library. We designed an English-like language and compiler to invoke these operations as well as those in the SL and to pass them parameters; and we designed a driver to execute all operations in a reliable, consistent manner.

Our primary objectives were:

- to define data types by the significance of their contents (date, phone, zip code, quantities, monetary amounts, etc.) and to perform data entry, validation, conversion and formatting automatically, regardless of storage type.
- to provide a very high-level English-like language that would be both easy to learn and self-documenting.
- to be able to use any number and type of files simultaneously including multiple databases, datasets, KSAM, printer, etc.
- to automatically store and load tables to supply values needed at run-time.
- to simplify declarations and eliminate the dull, boring redundant part of programming, where most errors are made
- to provide text specification syntax, including literal text, program variables and control characters, for use as program messages, report output, headings, etc.
- to work equally well for interactive and batch applications.

We wrote the system in SPL. It has been in daily operation for just over two years, during which time there have been two major rewrites and many additions and enhancements designed to further simplify its use and improve performance. Presently we are just putting the finishing touches on the final version which will incorporate all the things we have learned from these past two years of use and will reflect at every step what we feel will be the best design and coding available.

Since this is a new and unique approach to programming, there is no generic for it. We call it The Futura System, and it consists of a language, compiler, driver and an extensive procedure library. We have attempted to give it the ability to do anything, and when we have discovered something it would not do, we have added it. And while our primary intent was to use it for applications programming, we have found that it is eqnally strong and valuable as a powerful, versatile utility that is able to supplement and round-out the various system utilities quite handily.

Programming using FUTURA consists of Initialization Commands and Mainline Commands. The compiler reads and validates these, checks their parameters, provides default values where desirable, builds the Mainline binary command module, and formats and prints a program listing in one of several styles. The binary command module resides in the data stack and drives and controls the entire program execution.

InitCommands include:

STACK — which sets the total space the program will require. It has a default value of 3500 bytes, which will handle most utility needs as well as quite a lot of applications.

ALLOCATE — which dimensions the various buffers, should the defaults not be quite right.

BASE — used to open an IMAGE database.

SETS — for identifying the DataSets to be accessed.

FILE — for opening MPE and KSAM files.

TABLE — declares and loads a table, taking care of data conversion, statistics and storage automatically.

PRINTER — opens a printer file according to your specifications, including headlines, page numbers and location, forms-message, and all other parameters used with the line printer.

LOAD — which initializes any area in the data stack with any string or binary value.

INTEGERS — used to load a string of binary single-word integers at any location.

IDENTIFY — an InitCmmd that may be used or implied by the syntax, it sets up a table of identifiers for use throughout your program.

All InitCmmds that may be required must precede the Mainline.

Mainline Commands are names of logical operations such as ADD, MOVE, UPDATE(datatype), BINARY, etc. They may have up to five parameters, some of which are required and some optional. There are MainCmmds to do everything, and frequently there are several, giving the programmer meaningful options on how to accomplish a step. For interactive applications there are a dozen-or-so UPDATE (datatype) commands, such as UPDALFA, UPDNMBR, UPDZIP, UPDSSNO, etc., which not only accepts, validates, formats and displays, and stores the data, but also gives the programmer complete control and recognizes up to 8 special characters that permit backing up one or more fields, begin record over, check for mail, etc.

The fact that commands are the names of logical operations rather than language requirements means that when you have logically solved the problem you have also largely written the program.

Many MainCmmds return one or more values to the program such as the Condition Code, Length, DBStatus, Returned Value, etc. as may be needed.

Text strings to be used as prompts for interactive operations are passed automatically to the program, as the compiler counts them and stores them together with any control characters needed to handle the screen and make an eye-appealing presentation. Text needed for any other purpose is also passed, counted, stored, and recalled with little or no effort on the part of the programmer.

The MainCmmds themselves, the Identifiers, and the way the text strings are handled all provide a great deal of self-documentation right where it is needed in a program, and other documentation and comments may be added at any point. There is an index-building facility that produces an index for the documentation consisting of the program name and all of the comments in each program.

Many commands provide for testing and branching. They are processed uniformly by a subroutine, and branching may be either to a label or to another instruction. Subroutines may be nested up to 20 deep; they may call themselves, and they may reside anywhere in the Mainline. There are both Init and Mainline $INCLUDE commands, allowing routines to be stored separately where they may be used by several programs by including only the reference table.

The binary module together with any tables and initial values may be automatically saved and used again without recompiling by simply adding "$" to the STACK command ($STACK). This binary file may be purged any time changes are made, and it will be recompiled and saved at next compile if the "$" is in place.

In the handout pages we have included examples showing the program file as it is keyed using EDITOR, the normal program listing provided by the compiler which formats this Editor file and prints the permanent documentation, and a look as the terminal screen as each of these programs would appear when run, and a sample of the printer output where applicable.

These are some of the programs that were used to produce the Proceedings and the Exhibit and Conference Guide. While we asked that the papers for the Proceedings be keyed in cap and lower case using the EDITOR, with standard 72-byte records, the facts are that everyone used his/her own method — with record lengths from 60 to 160 bytes and some embedded control characters that would completely snarl our typesetting computer if they were not removed.

These are mostly small, simple programs that will illustrate the truly universal nature of the Futura System as a powerful and versatile utility. I have also brought the documentation for the Automatic TimeSharing Accounting and Billing System (ATSABS) which will show how it can be used for a large, complex system.

This will also show the automatic indexing and system documentation features that are available. We would be glad to have you all look this over and discuss it either at our booth or at other times and places by arrangement. This system totally automates our TimeSharing accounting and billing. It required approximately 5,000 lines of FUTURA code, and we estimate it would have required more than 30,000 of SPL.

Everything You Wanted to Know About Interfacing to the HP3000 PART I

Ross Scroggs
The Type Ahead Engine Company
Oakland, California

INTRODUCTION

It is important to realize that the information presented in this paper is my interpretation of the facts. The interpretation is not perfect, for surely I have included incorrect statements. If you believe that something here is incorrect, bring it to my attention. If I believe that you are wrong I will try to set you straight, but I will not argue about anything. I have included a list of references at the end of this paper from which I have obtained most of the information included here. If you desire to make all of your terminal attachments successful, obtain all of the references and read them. The most important piece of information I can give you is to start planning early when attaching terminals to the HP3000 and don't believe anything you read, if you haven't seen it work yourself, plan on having to solve a few problems. This paper is a guide to solving those problems, but it won't solve them for you.

Most of the experiments outlined in this paper were performed with the Bruno release of MPE-IV, I have subsequently been informed that the C release of MPE-IV fixed many terminal driver problems associated with the ADCC.

Asynchronous terminals are attached to the HP3000 Series I, II, and III through the Asynchronous Terminal Controller (ATC) and to the Series 30, 33, 40, and 44 through the Asynchronous Data Communications Controller (ADCC). This paper addresses issues involved in making a successful connection to one of these two devices. Terminals attach to the Series 64 through the Advanced Terminal Processor (ATP) which should make all of our lives simpler (though expensive) in the coming years. In its earlier versions the ATP will act much like the ATC in terms of interfacing to terminals. It features two major advances over the previous terminal controllers. First, there is a microprocessor controlling each terminal line, this removes considerable work from the CPU, the "character interrupt" problem. Second, the ATP can use either the RS-232 or RS-422 interface standards. RS-422 is a completely new electrical and mechanical interface that supports very high data rates over great distances with no errors, a typical example would be 9600 baud at 4000 feet. What this flexibility costs you is about $200 extra per terminal to provide a RS-232 to RS-422 adapter. These won't be required when terminals provide RS-422 interfaces.

Terminals attached to the ATC or ADCC are accessed primarily in two ways: as a session device or as a programmatically controlled device. A session device is one on which a user logs on with the HELLO or () commands and accesses the HP3000 through MPE commands. A programmatic device is one which is controlled by an application program that is run independently from the device. These two access methods are not mutually exclusive, a session device can be accessed programmatically and many MPE commands can be executed on behalf of a user who is accessing the system programmatically.

SESSION DEVICES

Attaching a terminal as a session device is typically the easier of the two methods. You must set the terminal speed, parity, subtype, and termtype correctly and provide the proper cable to complete the hookup.

Terminal Speed

The speeds supported by the ATC are 110, 150, 300, 600, 1200, and 2400 baud. The speeds supported by the ADCC are those of the ATC plus 4800 and 9600 baud. Unfortunately these two higher speeds can not be sensed by the ADCC and thus you must log on at a lower speed and use the MPE SPEED command to access the higher speed. (Use of subtype 4 and specifying any speed will allow a terminal to log on at that speed only, this includes 4800 and 9600. Note however, that if you use the :SPEED command the new speed specified will be required at your next logon.)

Terminal Parity

The format of characters processed by the HP3000 is a single start bit, seven data bits, a parity bit, and one stop bit (two at 110 baud). The parity bit may always be zero, always be one, computed for odd parity, or computed for even parity. Choosing the proper parity setting has been complicated by differences between the ATC and ADCC. The ATC inspects the parity bit of the initial carriage return received from the terminal and sets parity based on that bit. If the bit is a zero the ATC generates odd parity on output, if it is a one the ATC generates even parity on output. In either case the parity of incoming data is ignored and the parity bit is always set to zero before the data is passed to the requesting program. The ADCC also sets parity based on the parity bit of the initial carriage return but does so with a slight, but nasty twist. If the bit is a zero the ADCC passes through the parity bit supplied by the application program on output, if it is a one the ADCC generates even parity on output. If pass through parity was selected the parity of the incoming data is passed through to your program buffer. If even parity was selected the input data is checked for proper even parity. Thus, you should not use odd or force to one parity on the ADCC. The odd parity will be interpreted as pass through and the parity bits will wind up in your data buffer, string comparisons will fail because of the parity bits. Force to one parity will be interpreted as even and all input will cause parity errors.

Subtype

The ATC supports subtypes 0, 1, 2, 3, 4, 5, 6, 7, the ADCC support subtypes 0, 1, 2, 3, 4, 5. Subtypes 2, 3, 6, 7 concern half duplex modems and not me, so I will ignore them. Subtype 0 is the standard for directly attaching terminals without modems. (Note that terminals that are attached to multiplexors can fit in this category, the modem involved is managed by the multiplexor, not the HP3000.) Subtype 1 is the standard for attaching terminals that use full duplex modems such as Bell 103, 212 and Vadic 34xx. Both subtypes 0 and 1 speed sense on the initial carriage return. Subtype 4 is for direct attach terminals that will not be speed sensed, they will run at a fixed speed that is set at configuration time. This subtype is often used to prevent the HP3000 from trying to speed sense garbage, this sometimes occurs when using short-haul modems (line-drivers) that do not have a terminal attached to the other end. Subtype 5 is for modem attached terminals that will not be speed sensed.

Termtype

The ATC supports terminal types 0, 1, 2, 3, 4, 5, 6, 9, 10, 12, 13, 15, 16, 18, 19, 31, the ADCC supports terminal types 4, 6, 9, 10, 12, 13, 15, 16, 18, 19. Termtype 4 is for Datapoint 3300 terminals, it outputs a DC3 at the end of each output line and responds to backspace with a Control-Y, truly bizarre. (Termtype 4 on the ADCC does not output DC3s at the end of each line.) Termtype 6 is for low speed printers, it outputs a DC3 at the end of each line but responds to a backspace with a linefeed. (The linefeed is on the first backspace of a series, this allows you to type corrections under the incorrect characters.) Termtype 9 is the general purpose non-HP CRT terminal type. No DC3s are output at the end of the line (whew!!) and nothing strange happens on backspace, the cursor backs up just as you would expect. Termtype 10 is the standard for HP-26xx terminals. Termtype 13 is typically for those terminals at a great distance from the HP3000 for which some local intelligence echos characters and the 3000 should not. (Telenet and Tymnet charge you for those echoed characters, that's reason enough not to have the HP3000 echo them.) Termtypes 15 and 16 are for HP-263x printers. Termtype 18 is just like termtype 13 except that no DC1 is issued on a terminal read. Certain termtypes less than 10 specify a delay after carriage control characters are output to the terminal. The ATC handles this by delaying for the designated number of character times but does not output any characters. The ADCC actually outputs null characters. The most extreme case is termtype 6 which causes 45 nulls to be output after a cr/lf at 240 cps.

Cable

Direct attach terminals, subtypes 0 and 4, use only three signals in the cable: pin 2, Transmit Data, pin 3, Receive Data, and pin 7, Signal Ground. (Note that all signal names are given from the point of view of the terminal, not the modem or the HP3000 which acts like a modem.) Typically the cable will connect pin 2 at the terminal end to pin 2 at the HP3000, pin 3 at the terminal to pin 3 at the HP3000 and pin 7 at the terminal to pin 7 at the HP3000. This is not to say that your terminal does not require other signals, it just says that the HP3000 is not going to provide them, you must. If your terminal requires signals like Data Set Ready, Data Carrier Detect, or Clear To Send, you can usually supply these signals to the terminal with a simple cable patch. Jumper pin 4, Request To Send to pin 5, Clear To Send. Jumper pin 20, Data Terminal Ready to pin 6, Data Set Ready and pin 8, Data Carrier Detect. These two jumpers cause the terminal to supply its required signals to itself.

Modem attach terminals, subtypes 1 and 5, use seven signals in the cable: pin 2, Transmit Data; pin 3, Receive Data; pin 4, Request To Send; pin 6, Data Set Ready; pin 7, Signal Ground; pin 8, Data Carrier Detect; and pin 20, Data Terminal Ready. Naming the signals gets complicated since the HP3000 is acting like a modem and it is being attached to a modem. Typically, the cable that connects the HP3000 to the modem will connect pin 2 at the modem end to pin 3 at the HP3000, pin 3 at the modem to pin 2 at the HP3000, pin 4 at the modem to pin 8 at the HP3000, pin 6 at the modem to pin 20 at the HP3000, pin 7 at the modem to pin 7 at the HP3000, pin 8 at at the modem to pin 4 at the HP3000, and pin 20 at the modem to pin 6 at the HP3000.

The cable that attaches your terminal to a modem should be specified in your terminal owners manual, consult it for proper connections.

Flow Control

Flow control is the mechanism by which the speed/amount of data from the HP3000 to the terminal is controlled. The HP3000 supports two flow control methods, ENQ/ACK and XON/XOFF. The ENQ/ACK protocol is controlled by the system, after every 80 output characters the systems sends an ENQ to the terminal and suspends further output until and ACK is received back from the terminal. The suspension is of limited duration for termtypes 10 to 12, output resumes if no ACK is received in a short amount of time. The suspension is indefinite for termtypes 15 and 16, the ENQ is repeated every few seconds until an ACK is received. (It is the ENQ/ACK protocol that fouls up non-HP terminals that attempt to access the HP3000 through a port that is configured for an HP terminal. Most terminals do not respond to an ENQ with an ACK, you must do it manually by typing Control-F which is an ACK. An ENQ is generated by the HP3000 when the initial carriage return is received from the terminal, thus you get hung immediately. But, hit Control-F, and logon and specify the proper termtype in your HELLO command.)

The XON/XOFF flow control protocol is controlled by the terminal. When the terminal wishes to suspend output from the HP3000 it sends an XOFF (Control-S or DC3) to the HP3000 and sends an XON (Control-Q or DC1) to resume output. Unfortunately the HP3000 sometimes fails to properly handle one of the two characters and you either overflow your terminal or get hung up. This is particularly nasty when your terminal is a receive-only printer and you can't supply a missing XON. You're really dead if the HP3000 misses the XOFF. Termtype 13 has in my experience been the best termtype to use if your terminal requires the XON/XOFF flow control protocol. You can turn the echo back on with ESC :.

A special note on XON. If you inadvertently send an XON (DC1) to the HP3000 when output is not suspended, surprise you are now in paper tape mode and backspace, Control-X, and linefeed will act most strangely. Hit a single Control-Y to get out of this mode, the Control-Y will not be received by your program.

Some terminals perform flow control by raising and lowering a signal on their interface, the HP3000 can not handle this. You must either run the terminal at a low enough speed to avoid overflowing it or provide hardware to convert the high/low signal to ENQ/ACK or XON/XOFF, a costly affair.

A form of flow control used by HP terminals when inputting data to the HP3000 is the DC2/DC1 protocol. When the enter key is pressed on the terminal, a DC2 is sent to the HP3000 to alert it to a pending block mode transfer. When the HP3000 is ready to receive the data it sends a DC1 back to the terminal to start the data transfer. (Your program does not handle the DC2/DC1, but see below FCONTROL 28, 29.) This works fine except in certain circumstances. In certain modes the HP actually sends DC2 carriage return when the enter key is pressed. This is no problem unless the DC2 and CR do not arrive together. The CR may be seen as the end of the data if it comes sufficiently far behind the DC2, your program completes its request for data with nothing and the real data bites the dust when it finally shows up. The separation of the DC2 and CR can occur when using statistical multiplexors or when using Telenet or Tymnet. Be aware, this problem is infrequent, but unsettling when it occurs.

PROGRAMMATIC DEVICES

Attaching a terminal as a programmatic device is usually done when you want to attach a serial printer, instrument, data collection device, or other strange beast to the HP3000. An application program you write will typically control all access to the device, a user will not walk up to it, hit return, and log on. I will explain the various intrinsics that are used to access programmatic devices and will give short (incomplete) program segments that illustrate the access method.

Declarations

The following declarations will be assumed for all program segments shown.

```
begin

integer
    ilen,
    olen,
    pfnum:=0,
    pifnum:=0,
    pofnum:=0,
    precsize:=-256;   <<** pick a number large enough for the
                           maximum data transfer **>>

logical
    fcontrol'parm:=0,
    prev'echo;

logical array
    ibuff'(0:255),
    obuff'(0:255),
byte array
    pfname(0:7):="PROGDEV ";

byte array
    pdevice(0:7):="PROGDEV ";

byte array
    pifname(0:7):="IPROGDV ";

byte array
    pidevice(0:7):="IPROGDV ";

byte array
    pofname(0:7):="OPROGDV ";

byte array
    podevice(0:7):="OPROGDV ";

define
    fs'error'on'ccl= if <  then file'error(#,
    fs'error'on'ccne=if <> then file'error(#;

procedure print'message(enum);
  value
    enum;
  integer
    enum;
  option external;

intrinsic
    fclose,fcontrol,fopen,fread,fsetmode,fwrite,print'file'info,
    getprivmode,getusermode,iowait,terminate;

subroutine file'error(fnum,enum);
  value
    fnum,enum;
  integer
    fnum,enum;
  begin

    <<** simple file error handling subroutine, basic, not fancy
         or very good. **>>

    print'file'info(fnum);
    print'message(enum);  <<** supply something, but remember your
                               cliches, make it user-friendly! **>>
    terminate  <<** simple, direct, not too graceful **>>
  end;  <<* file'error *>>
```

FOPEN

You must call FOPEN to gain access to the device, I always use a formal file name to allow control of the open with file equations. If the device is unique in the system, I use its device name as the file name. The foptions specify CCTL, undefined length records, ASCII, and a new file. The aoptions specify exclusive access and input/output. Choose a record size that is larger than the maximum data transfer that will take place.

ATC — Opening a terminal with an HP termtype causes an initial ENQ to be output to the device on the first output, there must be an ACK reply from the device or your program will wait until the ENQ time-out occurs.

ADCC

For devices that are to be used exclusively in programmatic mode it is recommended that you REFUSE the device so that extraneous carriage returns from the device will not be speed sensed by the HP3000.

```
pfnum:=fopen(pfname,%604,%104,precsize,pdevice);
fs'error'on'ccl(pfnum,1);
```

FCLOSE

You call FCLOSE to release access to the device, some FCONTROL options exercised while the device was open are not reset by FCLOSE.

ATC — MPE sends a cr/lf to the device if it believes that the "carriage" is not at the beginning of the line, i.e., the last character output was not a linefeed.

ADCC — MPE sends a cr/lf to the device if it believes that the "carriage" is not at the beginning of the line, i.e., the last character output was not a linefeed or formfeed.

```
fclose(pfnum,0,0);
fs'error'on'ccl(pfnum,9);
pfnum:=0;  <<** I do this for error handling purposes **>>
```

FREAD

You call FREAD to get data from the device, many of the FCONTROL calls shown below affect how FREAD works. End-of-file is indicated by a record that contains ":EOF:". Any record with a colon in column one is an end-of-file to $STDIN, ":EOD", ":EOJ", ":JOB", ":DATA", and ":EOF:" are end-of-file to $STDINX. You should avoid linefeeds that follow carriage returns because garbage characters will be echoed to the terminal. (The inbound linefeed collides with the outbound linefeed coming as a result of the carriage return.)

```
ilen:=fread(pfnum,ibuff',precsize);
fs'error'on'ccl(pfnum,2);
if >
  then ; <<** handle eof **>>
```

You may want to trap certain errors returned by FREAD to your program: 22, software time-out; 31, end of line (alternate terminator); and 33, data lost.

ATC — The characters NULL, BS, LF, CR, DC1, DC3, CAN (Control-X), EM (Control-Y), ESC, and DEL are stripped from the input stream for both session and programmatic devices.

ADCC — The characters BS, LF, CR, CAN (Control-X), and EM (Control-Y) are stripped from the input stream for session devices. The characters BS, CR, and CAN (Control-X) are stripped from the input stream for programmatic devices.

The default parity cases are handled quite differently between the ATC and ADCC, you should exercise extreme caution when dealing with parity on the ADCC.

ATC — If the ATC is in the odd/out, no check/in mode all incoming characters have their parity bits set to zero. The same is true for even/out, no check/in mode.

ADCC — If the ADCC is in pass thru/out/in mode all incoming characters retain their parity bits, they are not set to zero. All special characters must have a zero parity bit to be recognized. If the ADCC is in even/out, check even/in mode the incoming characters must have proper even parity and their parity bits are set to zero. The second time you open this terminal the ADCC has switched to pass thru mode and all incoming characters retain their parity bits!!!

Each time you issue an FREAD to the terminal MPE sends a DC1 to the terminal to indicate that it is ready to accept data. Most devices ignore, totally, the DC1. If your a device reacts negatively to the DC1, use termtype 18 which suppresses the DC1 on terminal reads. The device must not send data to the HP3000 until it has received the DC1, otherwise the data will be lost. If the device does not wait for the DC1 you must supply external hardware that will provide buffering and wait for the DC1 or you can solve the problem on the HP3000 by using two ports to access the device. One port is opened for reading and the other for writing. A no-wait read is issued before the write that causes the device tc send data, then the read is completed.

```
getprivmode;  <<** necessary for nobuf, no-wait i/o **>>
pifnum:=fopen(pifname,%204,%4404,precsize,pidevice);
if <
  then begin
    getusermode;
    file'error(pifnum,1)
  end;
getusermode;
pofnum:=fopen(pofname,%604,%404,precsize,podevice);
fs'error'on'ccl(pofnum,1);
...
ilen:=fread(pifnum,ibuff',precsize);
fs'error'on'ccl(pifnum,2);
fwrite(pofnum,obuff',-olen,%cctl);
fs'error'on'ccne(pofnum,3);
iowait(pifnum,ibuff',ilen);
fs'error'on'ccne(pifnum,22);
```

When you attach your device to the two ports, connect pin 2, Transmit Data of the terminal to pin 2 of the read port, connect pin 3, Receive Data of the terminal to pin 3 of the write port, and pin 7, Signal Ground of the terminal to pin 7 of both ports. (This two port scheme was first introduced to me by Jack Armstrong and Martin Gorfinkel of LARC.)

FWRITE

You call FWRITE to send data to the device. The carriage control (cctl) value of %320 is often used to designate that MPE send no carriage control bytes, such as cr/lf, to the device. Some FCONTROL calls shown below affect how FWRITE works. Control returns to your program from FWRITE as soon as the data is loaded into the terminal buffers, it does not wait until all data has been output to the device.

ATC — Carriage control %61 is output as carriage return, formfeed (termtype 10).

ADCC — Carriage control %61 is output as formfeed (termtype 10).

The default parity cases are handled quite differently between the ATC and ADCC, you should exercise extreme caution when dealing with parity on the ADCC.

ATC — If the ATC is in odd/out mode all outgoing characters are given odd parity, even parity is generated when the mode is even/out. Simple.

ADCC — If the ADCC is in pass thru/out mode all outgoing characters retain their parity bits as passed to FWRITE. If the ADCC is in even/out mode all outgoing characters are given even parity. The second time you open this terminal the ADCC has switched to pass thru/out and all outgoing characters retain their parity bits!!!

```
    fwrite(pfnum,obuff',-olen,%cctl);
    fs'error'on'ccne(pfnum,3);
    <<** eof here is probably an error, I mean what is going on? **>>
```

FSETMODE — 4 — Suppress carriage return/linefeed

In normal operation a line feed is sent to the terminal if the input line terminates with a carriage return, a cr/lf is sent to the terminal if the line terminates by count, and nothing is sent if the line terminates with an alternate terminator. FSETMODE 4 suppresses these linefeeds and carriage returns. FSETMODE 0 returns to normal line termination handling, an FCLOSE also returns the device to the normal mode.

```
    fsetmode(pfnum,4);
    fs'error'on'ccl(pfnum,14);
```

FCONTROL

FCONTROL is the workhorse intrinsic for managing a programmatic device on the HP3000. Each use of FCONTROL which be shown separately but it will usually be the case that several calls will be used. Most calls are required only once, but the timer calls are required for each input operation. Each call will be identified by the controlcode parameter that is passed to FCONTROL.

FCONTROL – 4 – Set input time-out

This option sets a time limit on the next read from the terminal. It should always be used with devices that operate without an attached user to prevent a "hang." If something goes wrong with the device, your program will not wait forever, control will be returned to your program. The FREAD will fail and a call to FCHECK will return the errorcode 22, software time-out. No data is returned to your buffer in the case of a time-out, any data entered before the time-out is lost. If you issue a timeout for a block mode read the timer is stopped when the DC2 is received from the terminal, a new timer is then started which is independent of the timer set by this FCONTROL call. See the section below on enabling/disabling user block mode transfers.

```
    fcontrol'parm:=30;  <<** 30 second time-out **>>
    fcontrol(pfnum,4,fcontrol'parm);
    fs'error'on'ccl(pfnum,413);
    ilen:=fread(pfnum,ibuff',precsize);
    if <
      then begin
        fcheck(pfnum,errorcode);
        if errorcode <> 22
          then file'error(pfnum,errorcode*100+2);  <<** something else **>>
        <<** handle time-out **>>
      end;
```

FCONTROL – 10, 11 – Set terminal input/output speed

These FCONTROL options allow you to change the terminal input and output speeds. FCONTROL 37 can also be used to set terminal speed, it sets termtype as well and is the method that I prefer.

ATC — Split speeds are allowed.

ADCC — Split speeds are not allowed, FCONTROL 10 and 11 set both input and output speed.

FCONTROL – 12, 13 – Enable/disable input echo

These FCONTROL options allow you to enable and disable terminal input echoing. Many devices that attach to the HP3000 do not expect or desire echoing of the characters they transmit. This option along with FSETMODE 4 completely turns off input echoing. (Control-X is handled separately.) Echoing is not restored when a file is closed so you should always put echo back the way it was found.

```
fcontrol(pfnum,13,prev'echo);
fs'error'on'ccl(pfnum,1313);
...
<<** turn echo back on if it was previously on **>>
if prev'echo = 0
  then begin
    fcontrol(pfnum,12,prev'echo);
    fs'error'on'ccl(pfnum,1213)
  end;
```

FCONTROL – 14, 15 – Disable/enable system break

The break key is typically disabled when terrible things will happen if the user hits break and aborts out of a program. You, the programmer, always seem to need break for debugging purposes and discover that you have it turned off. System break can only be enabled for session devices, it is not allowed for programmatic devices. If break is entered on a session device the data already input will be retained and provided to the user program after a resume and the completion of the read. If a break is entered on a programmatic device a null will be echoed to the device but no data is lost.

FCONTROL – 16, 17 – Disable/enable subsystem break

Subsystem break is recognized only on session devices, it can be enabled on programmatic devices but has no effect. If a Control-Y is entered during a read, the read terminates and the data already input will be retained and provided to the user program after the Control-Y trap prodedure returns. If Control-Y is disabled any Control-Y will be stripped from the input but no trap procedure is called and the read continues. Control-Y trap procedures are armed by the XCONTRAP intrinsic. A subsystem break character other than Control-Y may be specified when unedited terminal mode (FCONTROL 41) is used.

ATC — In programmatic mode Control-Y's are always stripped from the input.

ADCC — In programmatic mode Control-Y is not stripped from the input if subsystem break is enabled.

FCONTROL – 18, 19 – Disable/enable tape mode

ATC — This is effectively an FSETMODE 4, an FCONTROL 35, and suppression of backspace echoing all rolled into one.

ADCC — Tape mode can not be enabled.

FCONTROL — 20, 21, 22 — Disable/enable terminal input timer, read timer

These options can be used to determine the length of time it took to satisfy a terminal read. It is not a timeout, that is FCONTROL 4. The manual states that you must enable the timer before each read so why is there a disable option? If you read the timer without enabling the timer, you get the time of the most recent read that did have the timer enabled. The number returned is the length of the read in one-hundreths of a second. Condition code > implies that the read exceeded 655.35 seconds.

```
fcontrol(pfnum,21,fcontrol'parm);
fs'error'on'ccl(pfnum,2113);
ilen:=fread(pfnum,ibuff',precsize);
fs'error'on'ccne(pfnum,2);
fcontrol(pfnum,22,fcontrol'parm);
fs'error'on'ccl(pfnum,2213);
```

FCONTROL – 23, 24 – Disable/enable parity checking

This option enables parity checking on input for the parity sense specified by FCONTROL 36. Parity checking is overridden by binary transfers (FCONTROL 27) or unedited mode (FCONTROL 41).

ATC — This option affects input parity checking only, output parity generation is controlled by FCONTROL 36.

ADCC — This options controls both input parity checking and output parity generation, FCONTROL 36 only specifies the type of parity.

FCONTROL – 25 – Define alternate line terminator

This option is used to select an alternate character that will terminate terminal input in addition to carriage return. It is useful if your device terminates input with something other than return.

ATC — Backspace, linefeed, carriage return, DC1, DC3, Control-X, Control-Y, NULL, and DEL are not allowed as terminators. The manual claims that DC2 and ESC are not allowed as terminators but they work. If a DC2 is the first input character from an HP termtype terminal the HP3000 drops the DC2 and sends a DC1 back to the terminal, it thinks a block mode transfer is starting. Any other DC2 is recognized as a terminator if enabled. By enabling user block mode transfers (FCONTROL 29) a DC2 as the first character will also be recognized as a terminator when enabled. For non-HP termtype terminals a DC2 is always recognized as a terminator when enabled.

ADCC — Backspace, linefeed, carriage return, Control-X, Control-Y, and NULL are not allowed as terminators. The manual claims that DC1, DC3, ESC, and DEL are not allowed as terminators, but they work. DC2 is allowed as a terminator but produces bizarre results unless unedited terminal mode (FCONTROL 41) is also enabled in which case the DC2 is recognized as a terminator in any position.

If a line terminates with an alternate terminator, it will be included in the input buffer and length and an error will be indicated for the read. You must call FCHECK to determine that the read terminated with the alternate character.

```
fcontrol'parm:=[8/0,8/"."];  <<** period is alternate terminator **>>
fcontrol(pfnum,25,fcontrol'parm);
fs'error'on'ccl(pfnum,2513);
...
ilen:=fread(pfnum,ibuff',precsize);
if <
  then begin
    fcheck(pfnum,errorcode);
    if errorcode <> 31
      then file'error(pfnum,errorcode*100+2);  <<** something else **>>
    <<** handle alternate terminator **>>
  end;
```

FCONTROL – 26, 27 – Disable/enable binary transfers

Binary transfers can be used to transmit full 8-bit characters to and from the terminal. On input a read will only be satisfied by inputting all characters requested, a carriage return or alternate terminator will not terminate the read. No cr/lf is echoed to the terminal at the end of the read. Thus, you must always know how many characters to read on each input from the terminal. Enabling binary transfers also turns off the ENQ/ACK flow control protocol and carriage control on output. No special characters are recognized on input. See the note under FCONTROL 25 about DC2 as the first input character on a line. If a session device is being accessed in binary mode, a break will remove the terminal from binary mode but it will not be returned to binary mode when a resume is executed.

FCONTROL – 28, 29 – Disable/enable user block mode transfers

As described above the normal sequence of events in a block mode transfer from an HP terminal to the 3000 is for the HP3000 to send a DC1 to the terminal indicating it readiness to accept data, the terminal sends a DC2 when the enter key is struck to indicate that it is ready to send data, the HP3000 responds with another DC1 when it is really ready to take the data, and the terminal sends the data. All of this is transparent to your program which just issues a big read. If your would like to participate in this handshake you enable user block mode transfers and MPE relinquishes control of the handshake. Your program would issue a small read, get the DC2, and issue another read to accept the data. This allows you to meddle around before the data shows up.

The terminal driver only supports block mode transfers with HP termtypes and performs one other function during block mode transfers. Normally you wouldn't put a timeout (FCONTROL 4) on a block mode read because the user can take an indefinite amount of time to fill a screen; but you would like to avoid terminal hangs because the block terminator from the terminal gets lost. This situation is handled by the driver for you, the portion of the read after the second DC1 is sent to the terminal is timed for (#chars in read/#chars per sec)+30 seconds. If the terminator is lost and the read times out, the read will fail and FCHECK will return error 27.

```
fcontrol(pfnum,29,fcontrol'parm);
fs'error'on'ccl(pfnum,2913);
...
ilen:=fread(pfnum,ibuff',-1);
fs'error'on'ccne(pfnum,2);
<<** meddle/muddle **>>
ilen:=fread(pfnum,ibuff',precsize);
fs'error'on'ccne(pfnum,2);
```

FCONTROL – 30, 31 – Disable/enable V/3000 driver control

This option is an undocumented option in which the terminal driver provides low level support for V/3000 use of terminals. When V/3000 issues a read to the terminal the driver outputs a DC1; the terminal user hits enter which causes a DC2 to be sent to the 3000; the driver responds with ESC c ESC H DC1 which locks the keyboard and homes the cursor; it appears that the driver also enables binary transfers because the second read only terminates by count, not by terminator. The portion of the read following the second DC1 is timed as described under FCONTROL 28, 29.

FCONTROL – 34, 35 – Disable/enable line deletion echo suppression

This option suppresses the !!!cr/lf echo whenever a Control-X is received from the terminal, the Control-X still deletes all data in the input buffer.

FCONTROL – 36 – Set parity

This FCONTROL option sets the sense of the parity generated on output and checked on input. The four possibilities are: 0, no parity, all 8 bits of the data are passed thru; 1, no parity, the parity bit is always set to one; 2, even/odd, even parity is generated if the original parity bit of the data was a zero, otherwise odd parity is generated; and 3, odd parity, odd parity is generated on all characters.

ATC — FCONTROL 36 sets the parity sense and enables output parity generation. FCONTROL 24 must be called to enable parity checking on input. An undocumented effect of this FCONTROL call is that the previous parity setting is returned in the controlcode parameter wiping out its original value!

ADCC — FCONTROL 36 sets the parity sense only. FCONTROL 24 must be called to enable output parity generation which results in input parity checking as well. An undocumented effect of this option is that the previous parity setting is returned in the controlcode parameter wiping out its original value!

Parity is not reset to the default case when a device is closed. This can be useful if you have a session device that can not run with the default parity. Each time the system is started run a program that opens the device, sets the parity, and closes the device. It can then be accessed as a session device with the required parity.

ATC — The following results were obtained when parity generation was enabled on output. All options performed as described in the manual.

ADCC — The following results were obtained when parity generation was enabled on output. Option 0, parity pass thru, resulted in even parity on all characters. Option 1, parity forced to one, resulted in odd parity on all characters. Option 2, even/odd parity, resulted in even parity on all characters regardless of the original parity bits of the characters. Option 3, odd parity, resulted in odd parity on all characters. Only option 3 performed as expected.

ATC, ADCC — The following results were recorded when parity checking was enabled on input. Option 0, parity pass thru, resulted in parity errors on all input except that with even parity. Option 1, parity forced to one, resulted in parity errors on all input except that with odd parity. Option 2, even/odd parity, resulted in parity errors on all input except that with even parity. Option 3, odd parity, resulted in parity errors on all input except that with odd parity. Options 2 and 3 performed as expected, options 0 and 1 did not. In all cases, parity bits are always set to zero before the data is passed to your program buffer.

HP has told me that the following is the parity story as of the C-delta version of MPE-IV.

ATC — Options 0 and 1 will not check parity on input, everything else as described above.

ADCC — Option 0 and 1 will be parity pass thru, everything else as described above.

FCONTROL – 37 – Allocate a terminal

In the old days you had to allocate a programmatic terminal before it could be used. Now you don't even though the manual claims that you do. This option is still useful because it allows you to set the termtype and terminal speed with one FCONTROL call. Common sense, mine at least, says to set termtype and speed each time a device is opened even if the proper values are configured in the i/o tables. Using this option allows use of a file equation redirecting the program to another device that might not be properly configured.

```
fcontrol'parm:=[11/speed,5/type];
fcontrol(pfnum,37,fcontrol'parm);
fs'error'on'ccl(pfnum,37);
```

FCONTROL – 38 – Set terminal type

This option allows you to set the terminal type, but use FCONTROL 37 and set type and speed all in one shot.

FCONTROL – 39 – Obtain terminal type information

Before changing the terminal type, get the current value and reset it when you are through.

FCONTROL – 40 – Obtain terminal output speed

Before changing the terminal speed, get the current value and reset it when you are through.

FCONTROL – 41 – Set unedited terminal mode

Unedited terminal mode is about the most useful FCONTROL option used to communicate with programmatic devices. It allows almost all control characters to pass through to the HP3000 but does not require reads of exact length as in binary transfers. Input will terminate on a carriage return or an alternate terminator if specified. The subsystem break character, replacing Control-Y, can also be specified.

ATC — Unedited terminal mode overrides input parity checking, no checking is performed and all input parity bits are set to zero. Output parity generation is performed normally.

ADCC — Unedited terminal mode processes parity in the same manner as edited mode, see the section on FREAD for an explanation.

Binary transfers enabled overrides unedited terminal mode enabled. If the input terminates with the end-of-record character or alternate terminator no cr/lf is sent to the terminal. If the input terminates by count a cr/lf is sent to the terminal unless an FSETMODE 4 has been done. Unedited mode does not turn off the ENQ/ACK flow control protocol on the ATC or ADCC. See the note under FCONTROL 25 about using DC2 as a terminator.

PTAPE

The manual describes PTAPE as the intrinsic to use to read paper tapes. (A fancy data-entry media that is becoming increasingly popular.) It can be used on the HP3000 to access devices that send up to 32767 characters all in one shot subject to a few limitations. The data must be record oriented with carriage returns between records, MPE will cut the data into 256 character records if there are no returns, and the whole mess must be terminated by a Control-Y. Certain buffering terminals allow you to fill their memory off-line, connect to a computer, and transmit all the data. This could save considerable time and money over dial-up phone lines.

DEBUGGING

If you have a requirement to attach a programmatic device to the HP3000 the worst strategy is to write some code on the 3000, plug the device in and start testing. Murphy says it won't work and it won't. The method I use is to test the device, then the code, and then the code and device together. I test the device by plugging it into an HP-2645 (or equivalent) terminal, turning on monitor mode, and simulate the HP3000 by typing on the keyboard. (Remember that you are hooking two terminals together, you will probably hook device pin 2 to 2645 pin 3, device pin 3 to 2645 pin 2, and device pin 7 to 2645 pin 7.) You can stimulate the device and observe all responses quite simply. Any strange behavior can be noted at this point. The next step is to write the code on the HP3000 to access the device in the manner determined by the first tests. Then plug the HP-2645, not the device, into the HP3000. Now type on the 2645 to simulate the device, continue until your code is debugged. Now you can plug the device into the HP3000 and you have a good (modulo Murphy) chance of actually getting it to work.

REFERENCES

Communications Handbook, Hewlett-Packard Company, April 1981 Part #30000-90105. This manual supersedes the HP Guidebook to Data Communications and the Data Communications Pocket Guide.

Don Van Pernis, "HP3000 Series II Asynchronous Terminal Controller Specifications," Computer Systems Communicator, #15, December 1977, page 2. This explains the terminal subtypes and signal requirements for the ATC.

Charles J. Villa, Jr., "Asynchronous Communications Protocols," Journal of the HP General Systems Users Group, Volume 1, #6, March/April 1978, page 2. Good introductory material.

John Beckett, "Poor Man's Multidrop," Journal of the HP General Systems Users Group, Volume 2, #1, May/June 1978, page 7. How to hook several terminals to the same port.

Tom Harbron, "Lightning, Transients and the RS-232 Interface," Journal of the HP General Systems Users Group, Volume III, #3, Third Quarter 1980, page 14. How to avoid being zapped.

MPE Intrinsics Manual, Hewlett-Packard Company, January 1981 Part #30000-90010. Chapter 5 discusses most of the FCONTROL options that are applicable in terms of the ATC, it is often inaccurate in describing the ADCC.

Everything You Wanted to Know About Interfacing to the HP3000 PART II

John J. Tibbetts
Vice President, Research & Development
The DATALEX Company

INTRODUCTION

The title of the this talk is "Everything You Wanted to Know about Interfacing to the HP3000-Part II." In Part I, Ross Scroggs described in great detail characteristics of the internals of the asynchronous communications protocol, especially for the benefit of those who would wish to tie foreign devices onto an HP3000 through the asynchronous port. This talk — the second part — is intended to take that discussion into a specific direction and discuss how, specifically, to connect intelligent devices, in particular microcomputers, to the asynchronous communications protocol of the HP3000. Note immediately that we are restricting our discussion of microcomputer communication to the asynchronous communications protocol. The reason for this is simply that most microcomputers are easily configurable to communicate asynchronously. Few microcomputer hardware and software packages have been assembled so far which will use bisynchronous communications protocol. Consequently, the reality of the current state of microcomputers suggests that asynchronous communications protocol will be the standard way of hooking up your microcomputer to the HP3000.

This talk will have two major parts. In the first part, we will discuss what is at issue in terms of features and capabilities in a remote communications program. We will discuss in some detail what our standard approaches to handling such capabilities as terminal emulation, sending and receiving files, simultaneously printing to a local printer, and control of the communications protocol from either the local end — that is, the microcomputer end — or from the remote computer end. In this talk we will refer to the local side as being the microcomputer and the remote side as being the remote or the host computer.

In the second part of the talk we will describe how you can actually get such a program running on your own machine. The choices are twofold: either buy one or write one. By using the criteria we have established in part one of the talk, we will try to outline some of the considerations of doing either of these.

One final note before we begin is that the remote communications capabilities tend to be a very hardware- and software-specific part of microcomputing. Whereas one can usually take a CPM program written in BASIC, for instance, and run it on most or all CPM implementations, one cannot expect to do the same with remote software. Remote software usually has to talk to pieces of your microcomputer which the operating system tends not to know anything about. During the course of the talk we will periodically make reference to a specific capability that is required for the remote program and in the published paper we will annotate them as a capability bullet that you will need to either have supplied to you or you will have to implement on your microcomputer to get this particular capability to be implemented for your remote program.

TERMINAL EMULATION

The first and perhaps the easiest capability to implement on your microcomputer is the emulation of a simple terminal.

- Capability — handling the remote port. Any of the operations we will be discussing for remote communications program presume that your microcomputer has a separate usable asynchronous communications port. Your program should be able to perform the following operations on that port:
  - read a character
  - write a character
  - test to see if a character is ready to be read.

  This last capability is the one that is usually missing in the standard microcomputing operating environments. In particular, CPM implements the read and write character routines as the reader and punch devices, respectively, but do not have a standard driver entry for testing the status of the remote port. Usually, you have to specifically write this capability for your own hardware if it hasn't been provided to you by someone else.

The standard procedure for implementing a terminal emulator is to write a polling loop. In the polling loop a very tight program loop tests to see if a character has been entered, either at the remote port or at the keyboard. If the character has been entered on either one, the character is then read and written to its opposite port. Thus, a character entered at the keyboard of the microcomputer, when sensed, would be written to the remote port and vice versa. It is important to make the polling loop as quick as possible. No code should be included in that loop unless it is absolutely necessary. Especially if you are writing in a high level language and especially if that language is interpretive, such as BASIC, you may have speed problems when trying to emulate a terminal at higher baud rates, say over 2400 baud. When writing in assembly language this is usually less important and you will find that you can support virtually any standard baud rate.

When emulating a half duplex terminal, any character entered at the keyboard should be immediately written back to the screen, thus providing the local echo. If terminal emulation requirements stopped here, a terminal emulator would be a very easy piece of software to write. Unfortunately, there are usually a few special problems with the terminal emulator.

The first problem is handling the break key. Many timesharing systems do not require break keys, but as any member of the HP3000 audience knows, the break key is a very crucial part of terminal handling on the HP3000. Unfortunately, the break key is not a character in the way any other keystroke on the terminal is. When depressed, the break key actually changes the electrical state of the transmit pin.

- Capability — sending a break. Most microcomputer communication ports have a mechanism by which the output port can be put into a break state. It almost always requires assembly language programming to implement a break key function. The actual reasoning behind break key handling goes beyond the scope of this talk. Suffice it to say that the preferred technique of break key transmission is, when the break key of a terminal is sensed, to put the output port into the break state until either 200 milliseconds have elapsed or a character comes in on the remote port.

Thus, to properly handle the break key our polling loop now needs to be expanded to test to see if the value entered from the keyboard is the break signal. When the break signal is sensed, the polling loop should then, instead of sending that character, invoke the send break routine to send a break.

The second capability which makes the terminal emulator more difficult relates to simultaneous printing of the terminal interactions on a printer which is hooked up to the microcomputer. The whole issue of printers, and especially printers that might hold up communications flow, is dealt with in a subsequent section.

SENDING AND RECEIVING FILES

Probably the main, useful work we would like for a communications program is to send files from our microcomputer to our HP3000 and receive files from the HP3000 down to the microcomputer. At first glance this may seem to be a rather simple operation. To send the file we should simply read the file from the local storage medium on the microcomputer and write it out the remote port. To receive a file we should simply read from the remote port and write to the diskette. If it were only so simple . . . There are three issues which will significantly complicate the issue of sending and receiving files. They are:

1. The vast majority of computers need time for themselves. What I mean by this statement is that at various times in the life of a computer it needs time to handle data it has been sending or receiving. On an HP3000, if you should try to type characters into it before it has put a prompt character up, you know that you will lose those characters. On a microcomputer, if you try to enter characters into most microcomputers while it is reading and writing a diskette file, for example, those characters will be lost. These phenomena reflect the fact that most computers are not designed to be able to handle communications of their terminal or remote ports at any time they are activated. A newer line of more commercially oriented microcomputers are beginning now to feature interrupt systems that do have full functioning typeahead systems which greatly ameliorate these problems. However, these microcomputers are definitely in the minority. Thus, our communications program needs to somehow be able to allow each computer to have time for itself when it needs it.

2. Communications lines tend to be rather noisy, especially if we are using the telephone system to transmit our data. Since file integrity is usually important, we need to come up with some kind of error checking protocol which can detect errors in the transmission of the data being sent or received. Interestingly enough, most of the programs running now on microcomputers for sending and receiving data do not handle error detection. The reason is that so far most microcomputer users who are using communications programs are doing so to make use of timesharing networks such as The Source for sending and receiving programs. As more, real, data processing functions, which might relate to shared databases or distributed data entry, are being built, clear data transmission protocols will become very important.

3. Most systems have some kind of difficulty with binary transmissions. This may not be a problem in applications in which only textual data needs to be sent. As time goes on, one finds the need to send binary data, for instance, to distribute object code of programs through the communications program. Thus it becomes desirable to be able to send binary files.

This is a summary of the problems — now let's take a look at some of the possible solutions.

MESSAGE HANDLING

Message handling is the general title by which we refer to the problem of the traffic control of the data being passed back and forth between the micro and the HP3000. The message handling protocol determines when data can safely be sent or received so that we never go faster than either of the machines can accommodate. The very first thing that becomes apparent after some investigation and experimentation is that the send and receive case are quite different from one another with *this* pair of computers. This is unusual when compared to communications software usually existing between microcomputer and microcomputer. In that case, the communications message handling is usually symmetric; that is, whatever convention is used to control data flow on the send side is also used symmetrically in the other direction to control it on the receive side. We have to do extra work on the HP3000 since none of it asynchronous communications protocol was designed for access by an intelligent terminal, and it ends up having some asymmetric properties which we have to deal with.

First, let's consider the case of sending data from a microcomputer to an HP3000. The first fact one must always be aware of when trying to send data to an HP3000 through its asynchronous communication port is that it can only read data from a device when a read is up; that is, when a read has been issued from a program. If you try to type ahead on an HP3000, the data is lost. Fortunately, in the HP3000 communications software a character is always sent whenever a read is put up. That character is the Control-Q or the DCl character. Thus, any device trying to send data to the HP3000 can simply wait until it sees a DCl and then send its record of data terminated by a carriage return. This type of *data interlocking* is the preferred method of sending data to an HP3000. For instance, this is the mechanism that LINK-125 uses in its protocol. It simply invokes FCOPY, and when FCOPY puts up its first read, it hands a record of data to it, terminates it with a carriage return, and proceeds with file transmission in that fashion.

But this is not always good enough. Consider this case: a message has been transfered to the HP3000, a carriage return sent following it, the HP3000 has issued another read, has sent the DCl back along the phone line and suddenly there is a noise burst on the phone line. The DCl coming back to the microcomputer is lost. The microcomputer is waiting there to transmit its next record of data with the DCl and then deadlocks because it never sees the DCl. This type of deadlocking is characteristic of trying to make too much out of a simple interlocking protocol. What we really find as more desirable is to write a communications program on the HP3000 which talks to the program on the microcomputer. This will allow the microcomputer program and the HP3000 program to issue reads with timeouts which would require that after a certain amount of time we give up on a particular read because of lost protocol characters or a dropped line. In the particular case of the missing DCl — and this is only one of the pathological conditions that can arise — the microcomputer program can simply, after a certain amount of time, send a message up the line which says something like, "Hey, are you still there?" to which the HP3000 program will response, "Yes, I am still here and here is another DCl" or may not respond at all if the machine has crashed or the line has gone down. This concept of using programs on both sides is really what differentiates very simple dumping of files up and down the line from more sophisticated communications protocols. I feel that this approach is required for any serious use of communications, especially with any bulk of data transmission which we would like to move reliably back and forth. Using this "program-to-program" approach, we can also perform some other more sophisticated error checking which we will get into shortly.

As we leave the send case, note this important fact. Make sure that after a record has been sent to the HP3000 with its carriage return the very next piece of work the microcomputer does is to turn around and wait for the DCl before it does anything else. One might be tempted to put up the next read to the diskette to pull the next record off while waiting for the DCl. If your microcomputer has the appropriate communications typeahead software on its remote port, you might be able to get away with this. However, in general the micrcomputer needs to wait for that interlock character to come back before it tries to do any other useful work. Otherwise, you will start missing DCls and your program will get hung up.

RECEIVING FILES

Just as we have done with the send case, let's examine the most trivial method of receiving files which would not rely on a program being run on the HP3000 side. The basic fact of life when receiving files is that the remote computer — the HP3000 — will be instructed to start sending down a file; perhaps we use FCOPY or the editor to start sending a file to us. The microcomputer is going to periodically need to write out the buffer it is accumulating to the disk drive. When it does this there will usually be a second in which it can't receive any data. The first approach is to just receive small files, in which case the microcomputer never writes out its data until it has collected the full file in memory. This of course limits the size of the file you can receive to the amount of available memory on the microcomputer, usually somewhere between 10,000 and 40,000 bytes. Obviously, this is an unsatisfactory method of receiving files unless your application is very limited. The next idea that comes to mind is making use of the X-on/X-off characteristics of the HP3000 to control this flow. As a human user, sometimes when a listing is coming out too quickly onto the CRT, we stop the flow by typing the

X-off key which is a Control-S and most of the time the HP3000 stops its transmission flow until you have done what you wanted and then you hit a Control-Q and the scrolling of the data output continues. Maybe we could have the microcomputer perform this function for us as a simple interlocking method.

The answer is that "Yes, we can," however, it is not the preferred method of receiving files. The reason for this is that, surprisingly, Control X-on/X-off protocol seems to have some holes in it on the HP3000 side. Someone told me that after some extensive testing they found that one out of five X-off characters seems to drop into a hole when sent to the HP3000. I have absolutely no way of verifying this other than to tell you it has happened a number of times to me. This doesn't make the use of this mechanism impossible, it simply complicates it somewhat.

Using this technique then, what you need to do is:

1. Build a large receive buffer.
2. Start receiving data until the buffer gets to 80% or 90% full.
3. Send an X-off character to the HP3000 but keep receiving the characters onto your microcomputer.
4. After some predetermined timeout time — perhaps a second of no characters coming in — assume it has finally absorbed the X-off character, and then you can proceed with your disk writes of the buffer.

But, if 4 or 5 characters have passed without stopping, send the X-off character again. Repeat these steps until the data transmission actually stops.

Just like the send file case, I recommend the use of a program on both sides. Using this technique we will simulate the kind of data interlocking protocol that the HP3000 uses. That is, every time the microcomputer is ready to issue a read to the remote HP3000 it will issue a character, perhaps for symmetry's sake a Control-Q, or any character of your choice. When that character is received by the program on the far side, that program will then send down the next record of data to the microcomputer followed by some standard termination character. After the message has been received, the microcomputer can set to work writing that message to the disk or doing whatever other housekeeping it would like to do. It then issues the next interlock character, and proceeds. This protocol also allows for the kind of timeout mechanisms that I described in the send case so that you can recover from lost transmission and especially lost protocol transmission. It will also easily accommodate the kind of error checking we will be talking about in the next section.

As always, there is a complication and a warning. Even in the case we have just described, we have not really built a symmetric communications protocol to the HP3000. The interlock character itself, which is going to be sent to the HP3000, has to be read by an HP3000 read. Of course, that HP3000 read will have a DCl coming right before it and any attempt to write the protocol character up the line before the HP3000 is able to read it results in a lost protocol character. Thus, some real world experimentation is usually needed in which some delay is required after the record has been received from the HP3000 so that it will have had time to finish writing the record and then put up the read which will read the next character interlock. It's for reasons like these, incidentally, that interfacing microcomputers to the HP3000 has not always been the simplest and the least frustrating of tasks.

The other item of note is that on reading characters into the microcomputer it is usually wise to strip out occurrences of the protocol characters that have accumulated in the asynchronous communication chip. These characters would be the line feed character which the HP3000 will usually tag onto the end of the carriage return unless you turn that off, and also the DCl character itself. Although these characters will be flying around during the transmision of the data, you don't want to include them into the data stream itself. They should be filtered out of the actual data flow.

ERROR HANDLING

Now that we have described the actual methods by which data can be sent and received, let's go on to the second defined problem in our data communications task — error handling.

The fact is that there are very few asynchronous communications protocols which go to this level, and I find this fact to be extremely regrettable. No serious large-scale interface of microcomputers to any kind of data processing network can be accomplished without real error checking. However, once we have built the proper send and receive frameworks with the right kinds of interlocking and assuming there is some intelligence and flexibility on both ends, it becomes rather easy to add the error handling phase. What are some of the usual techniques for adding error handling to the send and receive cases that we've described?

The standard mechanism, of course, is to add to each message sent or received some kind of check character or checksum which is used to check out the validity of the data. The simplest form of a checksum is an addition of the various character values of the message. For instance, if one record of data I am sending to the HP3000 is 40 characters long, the microcomputer can run through those 40 characters, add up the ASCII value of the 40 characters, and then produce a new character for the string and tag it onto the beginning or the end of that string. The HP3000 on the other end, when it has received the data, will go through the very same operation except that this time it will strip the character off and compare it with its own calculation of that string and see if they match.

There are some problems with the simple add-em-up checksum and there are many other sophisticated algorithms around — I can refer you to literally any book on communications for a description of CRC algorithms. The problem with CRC algorithm is that it's usually a fairly difficult algorithm to execute quickly enough on a microcomputer unless you are programming in an assembler language. The algorithm I have found to be very simple but very effective is an algorithm which adds and shifts the bits as the characters roll in. In this algorithm each character is added on to the checksum and then the checksum is multiplied by 2 which shifts all the bits to the higher order by one bit. Then it receives the next character and repeats the process. This final 16-bit quantity is then tagged onto the message.

You have to remember not to freely insert binary integers into the communications stream. Some adjustment of the value must be done when we are sending it to the HP3000 to make certain we are not sending a character it will have difficulty receiving.

Once a message has been sent to the other side with a checksum on it, the other side has the opportunity to examine that message and respond. The typical response is for the receiver to send back some predetermined character message which says either: the data was received successfully and you should proceed to the next block; or, alternatively, the data was not received correctly so retransmit the block just transmitted. I usually include a third state in this message traffic which indicates that something terrible has happened on one end or the other and to abort the entire transmission process altogether. You can include in this function the ability for the user to hit some kind of escape key and abort the communicatons traffic.

One other item I have found to increase reliability is to add sequence numbers on each of the messages sent or received. This would ensure that in some pathological case we don't actually get the blocks out of order; that is, in a case where an entire block has dropped out of the communications traffic. Although this is fairly rare, there are actually certain conditions which can cause something like that to happen. A sequence number which is checked on both sides for each block transmitted can protect against this possibility.

BINARY TRANSFERS

We have mentioned previously that it is generally unreliable to transmit 8-bit binary characters from a microcomputer up to an HP3000. What are the possible ways around this problem? The standard way is to simply convert the 8-bit binary traffic into hexadecimal strings, that is convert a binary character 255 into the ASCII string FF, etc. Of course, you would probably immediately see this means that there is a 50% reduction in communications efficiency. This technique is usually simple to perform and it is useful when the binary traffic is somewhat limited. A technique that I prefer is to translate seven 8-bit bytes into eight 7-bit bytes. This is quickly accomplished by gathering the 8th level bit of the 7 bytes input and building another byte and tagging it onto the back end of each 7-byte block. This effectively chops the 8th level off the communications stream at transmission time and is then reassembled on the far side. If this technique is used on the entire message, including checksums, sequence numbers or any kind of message identifier on the block, the whole communications interface becomes considerably simpler.

USING PRINTERS

It is often desirable in a communications protocol to log the data to the printer. For instance, on receiving a file to a microcomputer you may want to get a listing of it. Alternatively, you may wish, during terminal emulation, to get a copy of that session onto a hardcopy printer.

Like everything else mentioned in this talk, there are hidden catches. It seems simple enough to be able to put in a switch in the software — for instance, in the polling loop of the terminal emulator — that when a character has been sent or received, it should be sent to the printer port. However, many printers don't print at the communications speed. We will therefore distinguish between a fast printer and a slow printer. In this context, fast and slow do not have any absolute meaning to them. Fast means that the printer operates faster than the current communications context, and slow means that the printer operates slower than the current communications context. For example, in a 300-baud environment most printers (for example, an Epson matrix printer or a TI-810) will be fast printers. However, at 1200 baud most of the inexpensive matrix printers are slow printers, that is, they cannot keep up with the 1200-baud stream. Surprisingly, even printers such as the TI-810 which are rated at from 120 to 150 characters per second often cannot keep up with the 1200-baud flow of data. Therefore, the determination of whether a printer is a fast or slow printer can only be done by running a series of tests.

As you might now be able to suspect, there is very little difficulty with a true, fast printer in our communications program. Any characer we wish to print we simply output to the printer port. However, on a slow printer we have to do more resource balancing in that there is now another resource in the communications environment which needs time of its own. Adding a slow printer to a communications program can easily double the complexity of the communications environment.

At this point let me summarize a few of the major elements of printer integration:

• If the communications program has been implemented, as I have been suggesting, with a pair of programs on either end which have an interlocking mechanism, the simplest approach of integrating a slow printer is to print out the block of data during the time that the program is performing activities such as writing to the diskette or reading from the diskette. That is, after the message has been sent or received and before the interlock causes the pair of programs to proceed, the buffer of data sent or received can be put to the printer. An unfortuante side effect of this approach is that the printer is only printing between records. This does not take advantage of the fact that there may be sufficient time during the actual communications transmission to have the printer doing some useful work.

• Improvement on this scheme requires a new capability:

> Capability — Printer Ready — The printer ready capability says that our communications software can sense when the printer is available; that is, when a character can be written to the printer in such a way that the printer buffer will absorb the character instantly.

With a printer-ready capability in our software, we can build a more sophisticated operating environment in which we have, in effect, a small spooler being operated. That is, any data which has been successfully sent or received and is ready to print can be added to a print buffer. This buffer is metered out to the printer when the printer is ready. It is important that the remote communications facility always have top priority. The other mechanism that needs to be in effect in this type of environment is that as the printer buffer gets close to being full, a flag will go up which will hold the interlock the next time around until the print buffer has been totally cleared. Although this mechanism sounds somewhat obtuse, it actually provides a very effective method of integrating a slow printer into a communications environment.

• Integrating a slow printer into the terminal emulator mode can be accomplished by using the X-on/X-off character techniques I described in the receive section. That is, if printing is active, a mechanism will go into effect, during terminal emulation mode, such that the microcomputer dispatches X-on/X-off to control the characters coming from the remote computer into the microcomputer.

BIDIRECTIONAL CONTROL

The last major capability we will discuss in our communications program is the ability for the remote computer to assume control of the communications program. This can be very desirable in applications in which an operator may activate a communications program and get online with an HP3000 and perhaps start a UDC. At that point the UDC might take over all control of the microcomputer through the communications program such that it can request files to be sent and received. The following are a couple of points concerning bidirectional control:

• The basic concept in bidirectional control is that the remote computer can have some escape character which it can send to the microcomputer during terminal emulation that will cause the remote computer to assume command of the microcomputer. Commands can then be dispatched by the remote computer directly to the microcomputer. Be sure that all of the issues previously mentioned about interlocking and protocol are also supported by any direct interaction between the remote computer and the microcomputer.

• It is very useful to be able to have a capability whereby the remote computer can ask for directory listings directly from the microcomputer. This gives the remote computer a list of what files may need to be sent or received.

• You may wish to give the remote program the ability to actually terminate the communications session itself and to remove the user from the terminal emulator mechanism.

IMPLEMENTATIONS

The best thing you can do with communications software is to buy it rather than develop it. Unfortunately, this assumes that someone has developed the type of software running on the type of machine you desire. As we've indicated during the course of this talk, remote communications software tends to be more hardware dependent than almost any other software running on your computer. Not only is it hardware dependent, it is also operating system dependent. Thus, on a single machine — for instance, the Apple which can run the Apple DOS, the PASCAL operating system, and CPM (if the CPM card is added) — each of these three operating systems has a different file system and each has different requirements for its communications program. This means that no one program will solve all of your problems.

Let's consider some of the available implementations. Under CPM, there are a couple of programs fairly well known in the CPM community for doing file-to-file transfers. They are a program called CROSSTALK and a program called COMMX. Both of them are available through the major CPM software distributors. Both of the programs feature a non-protocol mode and a protocol mode. In the non-protocol mode you can easily make the software talk to your HP3000 by setting the DCl character to the interlock character. Unfortunately, on both of these programs the protocol mode which includes the checksumming algorithms is only usable when the program is talking to another CPM program of its own type. Clearly, these programs are written for CPM systems to talk to other CPM systems, not to another computer system. This means that if you do wish to turn one of these systems into a protocol checked operating environment, you need to do a little extra work on it. If you have an HP125 you can acquire LINK-125 which does a good, but not error-checked, link with the HP3000.

None of the programs I have seen feature bidirectional control which would allow, as I have described in the talk, the remote computer to assume the control of the microcomputer.

If you are running some variant of the UCSD PASCAL or UCSD p-System operating environment, then, with all due modesty, there is no better communications software available than that provided by our own company. It incorporates in a table-driven fashion, ready-to-run for the HP3000, all of the capabilities described in this talk, that is: full error-checked protocol, the ability to support fast and slow printers, full bidirectional control, and blank compression of the data. All of the software for communicating with an HP3000 has been worked out in great detail. Incidentally, we also support protocol-oriented communication for other p-Systems — that is, for p-System to p-System communication — as well as communication with the IBM 370 interactive operating system such as CMS, CSS, or TSO, and DEC-10, -11, and -20 support.

Something new is that the software distributors for the UCSD p-System now have a CPM file compatibility mode which, when available, will mean that we can also use our communications software to send and receive CPM files as well.

CONCLUSION

If there is any theme for a discussion of communications software, it is *"There is More Than Meets the Eye."* As I have repeatedly stressed, the very best way of solving your communication problems is finding someone else who has already solved them and acquire the software from them. This is my very strong recommendation when attempting to establish a remote communications network for your system.

I would also refer to the other talk I am giving at this meeting which encompasses distributed processing applications using microcomputers. It is entitled "Microcomputer-based Transaction Processing with Your HP3000" and it goes into some detail about the state of the art in microcomputer software for distributed processing.

Programming for Device Independence

John Hulme
Applied Cybernetics, Inc.
Los Gatos, California

INTRODUCTION

The purpose of this presentation is to discuss techniques and facilities which:

1. Isolate the programmer from specific hardware considerations
2. Provide for data and device independence
3. Allow the programmer to deal with a logical rather than a physical view of data and devices
4. Allow computer resources to be reconfigured, replaced, rearranged, reorganized, restructured or otherwise optimized either automatically by system utilities or explicitly by a system manager or databse administrator, without the need to rewrite programs.

The evolutionary development of these techniques will be reviewed from a historical perspective, and the specific principles identified will be applied to the problem of producing formatted screen applications which will run on any type of CRT.

WHAT IS A COMPUTER?

As you already know, a computer consists of one or more electronic and/or electromechnical devices, each capable of executing a limited set of explicit commands. For each type of device some means is provided to allow the device to receive electrical impulses indicating the sequence of commands it is to execute. In addition to commands, most of these devices can receive electrical impulses representing bits of information (commonly called data) which the device is to process in some way. Nearly all of these devices also produce electrical impulses as output, which may in turn be received as commands and/or data by other devices in the system.

Nowadays, most devices also have some form of "memory" or storage media where commands or other data can be recorded, either temporarily or semi-permanently, and a means by which that data can later be received in the form of electrical impulses.

The tangible, visible, material components which these devices are physically made up of is generally called computer *hardware*. Any systematic set of instructions describing a useful sequence of commands for the computer to execute can be called computer *software*. As we will see later, software can be further subdivided into *system software*, which is essentially an extension of the capabilities of the hardware, and *application programs*, which instruct the computer how to solve specific problems, handle day-to-day applications, and produce specific results.

Originally it was necessary for a computer operator to directly input the precise sequence of electrical signals by setting a series of switches and turning on the current. This process was repeated over and over until the desired sequence of instructions had been executed.

By comparison with today's methods of operating computers, those earlier methods can truly be called archaic. Yet the progressive advancement of computer systems from that day to this, however spectacular, is nothing more than a step-by-step development of hardware and software building blocks, an evolutionary process occurring almost entirely during the past 25 years.

ENGINEERING AND AUTOMATION

I think we mostly take for granted the tremendous computing power that is at our fingertips today. How many of us, before running a program on the computer, sit down and think about the details of hardware and software that make it all possible? For that matter, who stops to figure out where the electrical power is coming from before turning on a light or using a household appliance? Before driving a car or riding in an airplane, who stops to analyze how it is put together?

Probably none of us do, and that is exactly what the design engineers intended. You see, it is the function of product engineering to build products which people will buy and use, which usually means building products which are easy to use. The fact that we don't have to think about how something works is a measure of how simple it is to use.

Wherever a process can be automated and incorporated into the product, there is that much less that the consumer has to do himself. Instead of cranking the engine of a car, we just turn a key. Instead of walking up 30 flights of stairs, we just push a button in the elevator.

It's not that we are interested in being lazy. We are interested in labor-saving devices because we can no longer afford to waste the time; we have to meet deadlines; we want to be more efficient; we want to cut costs; we want to increase productivity. We also want to reduce the chance for human error. By automating a complicated process, we produce consistent results, and when those results are thoroughly debugged, error is virtually eliminated. We can rely on those consistent results, which sometimes have to be executed with split second timing and absolute accuracy. Without reliable results there might be significant economic loss or danger to life and limb. Imagine trying to fly modern aircraft without automated procedures.

Automation also facilitates standardization, which allows interchangeability of individual components. This leads to functional specialization of components, which in turn leads to specialization of personnel, with the attendant savings in training and maintenance costs. And because the engineering problem only has to be solved once, with the benefits to be realized every time the device is used, more time can profitably be spent coming up with the optimum design.

BUILDING BLOCKS

In my opinion, the overwhelming advantage of automating a complicated process is that the process can thereafter be treated as a single unit, a "black box," if you will, in constructing solutions to even more complicated processes.

Later, someone could devise a better version of the black box, and as long as the functional parameters remain the same, the component could be integrated into the total system at any time in place of the original without destroying the integrity of any other components.

It is this "building-block" approach which has permitted such remarkable progress in the development of computer hardware and software. As we review the evolution of these hardware/software building blocks, keep in mind that the chronological sequence of these developments undoubtedly varied from vendor to vendor as a function of how each perceived the market demand and how their respective engineering efforts progressed.

ONE STEP AT A TIME

Even before the advent of electronic computers, various mechanical and electro-mechanical devices had been produced, some utilizing punched card input. Besides providing an effective means of input, punched cards and paper tape represent a *rudimentary storage medium.* Incorporating *paper tape and card readers* into early computer systems not only allowed the user to input programs and data more quickly, more easily, and more accurately (compared with flipping switches manually), but on top of that it allowed him to enter the same programs and data time after time with hardly more effort than entering it once.

The next useful development was the *"stored program"* concept. Instead of re-entering the program with each new set of data, the program could be read in once, stored in memory, and used over and over.

This concept is an essential feature of all real computers, but it would have been practically worthless except for one other essential feature of computers known as *internal logic.* We take these two features so much for granted that it's hard to imagine a computer without them. In fact, without internal logic, computers really wouldn't be much good for anything, since they would only be able to execute a program in sequential order beginning with the first instruction and ending with the nth. Internal logic is based on special hardware commands which provide the ability first of all to test for various conditions and secondly to specify which command will be executed next, depending on the results of the test. In modern computer languages, internal logic is manifest in such constructs as IF statements, GO TO statements, FOR loops, and subroutine calls.

But at the stage we are discussing there were no modern programming languages, just the language of electrical signals. These came to be represented as numbers (even letters and other symbols were given a numeric equivalent) and programs consisted of a long list of numbers.

Suppose, for example, that the numbers 17, 11, and 14 represented hardware commands for reading a number, adding another number to it, and storing the result, respectively, and suppose further that variables A through Z were stored in memory locations 1 through 26. Then the program steps to accomplish the statement "give Z a value equal to the sum of X and Y" might be expressed as the following series of numbers, which we will call *machine instructions:*

17, 24, 11, 25, 14, 26

In essence, the programmer was expected to learn the language of the computer.

A slight improvement was realized when someone thought to devise a meaningful mnemonic for each hardware command and to have the programmer write programs using the easier-to-remember mnemonics, as follows:

READ, 24, ADD, 25, STORE, 26

or perhaps even

READ, X, ADD, Y, STORE, Z.

After the programmer had described the logic in this way, any program could be readily converted to the numeric form by a competent secretary. But since the conversion was relatively straightforward, it would be automated, saving the secretary some very boring work. A special computer program was written, known as a *translator.* The mnemonic form, or *source program* as it was known, was submitted as input data to the translator, which substituted for each mnemonic the equivalent hardware command or memory location, thus producing machine instructions, also known as *object code.* Translators required two phases of execution, or two passes, one to process the source program and a second to execute the resulting object code. Once the program functioned properly, of course, it could be executed repeatedly without the translation phase.

It would have been possible for the hardware engineers to keep designing more and more complicated hardware commands, and to some extent this has been done, either by combining existing circuitry or by designing new circuits to implement some new elemental command. Each new machine produced in this way would thus be more powerful than the last, but it would have been economically prohibitive to continue this type of development for very long and the resulting machines would have been too large to be practical anyway.

Engineers quickly recognized that instead of creating a more powerful command by combining the circuitry of existing commands, the equivalent result could be achieved by combining the appropriate collection of commands in a miniature program. This mini-program could then be repeated as needed within an application program in place of the more complex command. Or better yet, it could be kept at a fixed location in memory and be accessed as a subroutine just the same as if it were actually a part of each program.

Another approach was to use an *interpreter,* a special purpose computer program similar to a translator. The interpreter would accept a source program in much the same way as the translator did, but instead of converting the whole thing to an object program, it would cause each hardware command to be executed as soon as it had been decoded.

Besides requiring only one pass, interpreters had the added advantage of only having to decode the commands that were actually used, though this might also be a disadvantage, since a command used more than once would also have to be decoded more than once.

The chief benefit of an interpreter lay in its ability to accept mnemonics for commands more complex than those actually available in the hardware, and to simulate the execution of those complex commands through the use of subroutines. In this way, new commands could be implemented without any hardware modifications merely by including the appropriate subroutines in the interpreter. This step marked the beginning of system software.

In addition, source programs for nearly any computer could be interpreted on nearly any other computer, as long as someone had taken the time to write the necessary interpreter. Interpreters could even be written for fictional computers or computers that had been designed but not yet manufactured. This technique, though generally regarded as very inefficient, provided the first means of making a program transportable from one computer to another incompatible computer.

It is possible, of course, to apply this technique to translators as well, allowing a given mnemonic to represent a whole series of commands or a subroutine call rather than a single hardware instruction. Such mnemonics, sometimes called macros, gave users the impression that the hardware contained a much broader repertoire of commands than was actually the case.

Implementing a new feature in software is theoretically equivalent to implementing the same function in hardware. The choice is strictly an economic one and as conditions change so might the choices. One factor is the universality or frequency with which the feature is likely to be used. Putting it in hardware generally provides more efficient execution, but putting it in the software is considerably easier and provides much greater flexibility.

The practice of restricting hardware implementation to the bare essentials also facilitated *hardware standardization* and compatibility, which was crucial to the commercial user who wanted to minimize the impact on all his programs if he should find it necessary to convert to a machine with greater capacity. Beginning with the IBM 360 series in 1964 *"families" of compatible hardware* emerged, including the RCA Spectra 70 series, NCR Century series, and Honeywell 200 series, among others.

Each family of machines had its own *operating system,* software monitor, or executive system overseeing the operation of every other program running on the machine. In some systems, concurrent users were allowed, utilizing such techniques as memory partitioning, time-sharing, multi-threading, and memory-swapping. Some form of *job control language* was devised for each operating system to allow the person submitting the jobs to communicate with the monitor about the jobs to be executed.

Introducing families of hardware did not solve the problem of compatibility between one vendor and the next, however, a problem which could only be solved by developing programming languages which were truly independent of any particular piece of hardware.

Since the inventors of these so-called *higher-level languages* were not bound by any hardware constraints, an effort was made to make the languages as natural as possible. FORTRAN imitate the language of mathematical formulas, while ALGOL claimed to be the ideal language for describing algorithmic logic; COBOL provided an English-like syntax, and so on.

Instead of having to learn the computer's language, a programmer could now deal with computers that understood his language. Actually, it was not the hardware which could understand his language, but a more sophisticated type of translator-interpreter known as a *compiler.*

To the degree that a particular language enjoyed enough popular support to convince multiple vendors to implement it, programs written in that language could be transported among those machines for which the corresponding compiler was available.

The term compiler may have been coined to indicate that program units were collected from various sources besides the source program itself, and were compiled into a single functioning module. Subroutines to perform a complex calculation such as a square root, for example, might be inserted by the compiler whenever one or more square root operations had been specified in the body of the source program.

Embedding subroutines in the object code was not the only solution, however. It became more and more common to have the generated object programs merely "CALL" on subroutines which were external to the object program, having been pre-compiled and stored in vendor-supplied *"subroutine libraries."* This concept was later extended to allow users a means of placing their own separately-compiled modules in the library and accessing them wherever needed in a program.

I should mention that an important objective of any higher level language should be to enable a user to describe the problem he is solving as clearly and concisely as possible. Although the emphais is ostensibly on making the program easy to write, being able to understand the program once it has been written may be an even greater benefit, particularly when program maintenance is likely to be performed by someone other than the original author.

It is well-known that program maintenance occupies a great deal of the available time in the typical data processing shop. Some studies estimate the figure at over 50% and increasing. In order to be responsive to changing user requirements, it is essential to develop methods which facilitate rapid and even frequent program changes without jeopardizing the integrity of the system, and without tying up the whole DP staff.

To avoid having to re-debug the logic every time a change is made, it is often possible to use *data-driven or table-driven programming techniques.* The portion of the program which is likely to change, and which does not really affect the overall procedural logic of the program, is built into tables or special data files. These are accessed by the procedural code to determine the effective instructions to execute.

The most common example in the United States, and perhaps in other countries as well, is probably the table of income tax rates, which changes by law now at least once a year. The algorithm to compute the taxes changes very rarely, if at all, so it does not have to be debugged each time the tables change. In simple cases like this, non-programmer clerks might safely be permitted to revise the table entries.

In more sophisticated applications, tables of data called *logic tables* may more directly determine the logic flow within a program. The program becomes a kind of interpreter, and elements in the logic table may be regarded as instructions in some esoteric machine language. Such programs are generally more difficult to thoroughly debug, but once debugged provide solutions to a broad class of problems without ever having to revise the procedural portion of the program.

Sometimes, logic-controlling information is neither compiled into the program nor stored in tables, but is provided to the program when it is first initiated or even during the course of execution, in the form of *run-time parameters* or *user responses.* The program has to be pre-programmed to handle every valid parameter, of course, and to gracefully reject the invalid ones, but this method is useful for cutting down the number of separate programs that have to be written, debugged, and maintained. For example, why write eight slightly different inventory print programs, if a single program could handle eight separate formats through the use of run-time options?

Incidentally, program recompilations need not always cause alarm. Through the proper use of *COPY code,* programs can be modified, recompiled, and produce the new results without the original source program ever having to be revised. This is made possible by a facility which allows the source program to contain references to named program elements stored in a COPY library instead of having those elements actually duplicated within the program. A COPY statement is in effect a kind of macro which the compiler expands at the time it reads in the source program.

For example, if a record description or a table of values appears in one program, it is likely to appear in other programs as well. It is faster, easier, safer, and more concise to say "COPY RECORD-A." or "COPY TABLEXYZ." than to re-enter the same information again and again. And if for some reason the record layout or table of values should have to be changed, merely change it in the COPY library, not in every program.

By changing the contents of a COPY member in this way and subsequently recompiling selected programs in which the member is referenced, those programs can be updated without any need to modify the source. If procedure code is involved, the new COPY code only need be debugged and retested once rather than revalidating all the individual programs.

Where blocks of procedural code appearing in many programs can be isolated and separately compiled, however, this would probably be better than using COPY code. For one thing, the *separate modules* would not have to be recompiled every time the procedural code was revised.

BITE-SIZE PIECES

Breaking a complex problem into manageable independent pieces and dealing with them as separate problems is a valuable strategy in any problem-solving situation. Such a strategy has added benefits in a programming environment:

1. Smaller modules are typically easier to understand, debug, and optimize.
2. Smaller modules are usually easier to rewrite or replace if necessary.
3. Independent functions which are useful to one application are often useful to another application;

using an existing module for additional applications cuts down on programming, debugging, and compilation time.

4. Allowing applications to share a module reduces memory requirements.
5. Having only one copy of a module ensures that the module can be replaced with a new version from time to time without having to worry that an undiscovered copy of an older version might still be lurking around somewhere in the system.

The fact that a routine only has to be coded once usually more than compensates for the extra effort that may have to go into generalizing the routine. The more often it's used, the more time you can afford to spend improving it.

SYSTEM SOFTWARE

Functions which are so general as to be of value to every user of the computer, such as I/O routines, sort utilities, file systems, and a whole host of other utilities, are usually included in the system software supplied by the hardware vendor. Just what facilities are provided, how sophisticated those facilities are, and whether the vendor Charges anything extra for them, is a matter of perceived user need and marketing strategy. Sometimes vendors choose to provide text editors and other development tools, and sometimes they don't. Sometimes they provide a very powerful database management system, sometime only rudimentary file access commands. And so on.

When hardware vendors fail to provide some needed piece of software, it may be worthwhile for the user to write it himself. If the need is general enough, software vendors may rush in to fill the void; or perhaps user pressure will eventually convince hardware vendors to implement it themselves.

In this way, many alternative products may become available, and the user will have to evaluae which approach he wishes to take advantage of, based on such factors as cost, efficiency, other performance criteria, flexibility of operation, compatibility with existing software, and the comparative benefits of using each product.

PRINCIPLES OF GOOD SYSTEM DESIGN

In case you may need to design your own supporting software, or evaluate some that is commercially available, let's summarize the techniques which will permit you to achieve the greatest degree of data, program, and device independence. I have already given illustrations of most of the following principles:

1. *Modularity* — Conceptually break everything up into the smallest modules you feel comfortable dealing with.

2. *Factoring* — Whenever a functional unit appears in more than one location, investigate whether it is feasible to "factor it out" as a separate module (this is analogous to rewriting A*B+A*C+A*D as A*(B+C+D) in math).

3. *Critical Sections* — Refrain from separating modules which are intricately interconnected or subdividing existing modules which are logically intact.

4. *Independence* — Strive to make every module self-contained and independent of every external factor except as represented by predefined parameters.

5. *Interfacing* — Keep to a minimum the amount of communication required between modules; provide a consistent method of passing parameters; make the interface sufficiently general to allow for later extensions.

6. *Isolation* — Isolate all but the lowest-level modules from all hardware considerations and physical data characteristics.

7. *Testing* — Test each individual module by itself as soon as it is completed and as it is integrated with other modules.

8. *Generalization* — Produce modules which solve the problem in a general way instead of dealing with specific cases. Be careful, however, not to overgeneralize. Trying to make a new technology fit the mold of an existing one may seem like the best modular approach, and the easiest to implement, but the very features for which the new technology has been introduced must not become lost in the process.

EXAMPLE — When CRTs were first attached to computers they were treated as teletypes, a class of I/O devices incompatible with two of the CRT's most useful features: cursor-addressing and the ability to type over existing characters. Putting the CRT in block-mode and treating it as a fixed-length file represents the opposite extreme: the interactive capalities are suppressed and the CRT becomes little more than a batch input device, a super-card-reader in effect.

9. *Standardization* — Develop a set of sound programming standards including structured programming methods, and insist that each module be coded in strict compliance with those standards.

10. *Evaluation* — Once the functional characteristics have been achieved, use available performance measurement methods to determine the areas which most need to be further optimized.

11. *Piecewise Refinement* — Continue to make improvements, one module at a time, concentrating on those with the largest potential for improving system performance, user acceptance, and/or functional capabilities.

12. *Binding* — For greater flexibility and independence, postpone binding of variables; for greater efficiency of execution, do the opposite; pre-bind constants at the earliest possible stage.

BINDING

As the name suggests, "binding" is the process of tying together all the various elements which make up an executing program. Binding occurs in several different stages ultimately making procedures and data accessible to one another.

For example, the various statements in an application program are bound together in an object module when the source program is compiled. Similarly, the various data items comprising an IMAGE database become bound into a fixed structure when the root file is created. A third case of binding involves the passing of parameters between separately compiled modules.

Remember that at the hardware level, where everything is actually accomplished, individual instructions refer to data elements and to other instructions by their location in meory. The "address" of these elements must either be built into the object code at the time a program is compiled, be placed there sometime *prior* to execution, or be provided *during* execution. Likewise, information governing the flow of logic can be built into the program originally, placed in a file which the program accesses, passed as a parameter when the program is initiated, or provided through user interaction during execution.

Binding sets in concrete a particular choice of options to the exclusion of all other alternatives. Delayed binding therefore provides more flexibility, while early binding provides greater efficiency. Binding during execution time can be especially powerful but at the same time potentially critical to system performance. In general, variables should be bound as early as possible unless you specifically plan to take advantage of leaving them unbound, in which case you should delay binding as long as it proves beneficial and can still be afforded. Incidentally, on the HP3000, address resolution between separately-compiled modules will occur during program preparation (PREP) except for routines in the segmented library, which will be resolved in connection with program initiation. If your program pauses initially each time you run it, this run-time binding is the probable cause.

A SPECIFIC APPLICATION

About five years ago, we were faced with the problem of developing a system of about 300 on-line application programs for a client with no previous computer experience. Their objective was to completely automate all record-keeping, paper-flow, analysis, and decision making, from sales and engineering to inventory and manufacturing to payroll and accounting. The client had ordered an HP3000 with 256K bytes of memory and had already purchased about 20 Lear-Sigler ADM-1 CRTs. About 12 terminals were to be in use during normal business hours for continuous interactive data entry; the remaining eight terminals were primarily intended for inquiry and remote reporting. Up-to-date information had to be on-line at all times using formatted screens at every work station. Operator satisfaction was also a high priority, with two- to five-second response time considered intolerable.

DISCUSSION QUESTIONS

Based on the "principles of good system design" summarized earlier, what recommendations would you have made to the development team?

At the time, HP's Data Entry Language (DEL) seemed to be the only formatted screen handler available on the HP3000. Consultation with DEL users convinced us it was rather awkward to use and exhibited very poor response time. Also it did not support non-HP character-mode terminals.

We elected to write a simple character-mode terminal interface, which was soon expanded to provide internal editing of data fields, and later enhanced to handle background forms. We presently market this product under the name TERMINAL/3000. You've probably heard of it.

The compact SPL routines reside in the system SL and are shared by all programs. The subroutine which interfaces directly with the terminals is table-driven to ensure device-independence. By implementing additional tables of escape sequences, we have added support for more than a dozen different types of terminals besides the original ADM-1's.

If we were faced with a similar task today, would your recommendations be any different?

After completing most of the project, we did what should have been done much earlier: we implemented a CRT forms editor and COBOL program generator which together automate the process of writing formatted-screen data entry programs utilizing TERMINAL/3000. We call this approach "results-oriented systems development"; the package is called ADEPT/3000. Programs which previously took a week to develop can now be produced in only half a day.

Since we were using computers to eliminate monotonous tasks and improve productivity for our clients, it was only natural that we should consider using computers to reduce monotony and increase productivity in our own business, the business of writing application programs. If you write application programs or manage people who do, you also may wish to take advantage of this approach.

What features of VIEW/3000 would have made it unsuitable for this particular situation?

- not available five years ago
- HP2640 series of terminals only
- block-mode only (not interactive field-by-field)
- requires huge buffers (not enough memory available)
- response time and overall system performance inadequate

From what you know of TERMINAL/3000 and ADEPT/3000, how do these products enable a programmer to conform to the principles of good system design?

TERMINAL/3000 itself: modular, well-factored, single critical section, device-independent, independent of external formats, simple 1-parameter interface, table-driven hardware isolation, well-tested, generalized, optimized for efficiency, run-time binding of cursor-positioning and edit characteristics.

ADEPT/3000: produces COBOL source programs that are modular, well-segmented, device-independent, and contain pre-debugged logic conforming to user-tailored programming standards; built-in interfaces to TERMINAL/3000 and IMAGE/3000 (or KSAM/3000) isolate the programs from hardware considerations and provide device and data independence.

BIBLIOGRAPHY

Boyes, Rodney L., *Introduction to Electronic Computing: A Management Approach* (New York: John Wiley and Sons, Inc., 1971).

Hellerman, Herbert, *Digital Computer System Principles* (New York: McGraw-Hill Book Co., Inc., 1967).

Knuth, Donald E., *The Art of Computer Programming* (Reading, Mass.: Addison-Wesley Publishing Company, 1968).

Swallow, Kenneth P., *Elements of Computer Programming* (New York: Holt, Rinhart and Winston, Inc., 1965).

Weiss, Eric A. (ed.), *Computer Usage Fundamentals* (New York: McGraw-Hill Book Co., Inc., 1969).

Selecting Application Software and Software Suppliers

Steven J. Dennis
Smith, Dennis & Gaylord

We begin by looking at some of the data regarding the assumptions that have evolved regarding standard software packages and the software package industry.

THE MYTHS

Following are some of the myths — beliefs ("beliefs" are assumptions which may or may not reflect reality) — which have evolved over the past decade or so of application packages evolution:

• *Myth:* We can safely go ahead with our hardware purchase since there MUST be plenty of good software systems available.

• *Fact:* There are surprisingly few firms nationally that have developed truly viable packages. And fewer still that will be able to meet YOUR requirements.

In fact of the literally, tens of thousands of software firms, barely a dozen have sales of more than $10,000,000 annually. And of these, the largest barely tops $35 million . . . hardly international giants!

• *Myth:* Since all packages are essentially the same, we will budget for the low priced one . . . that will help keep the costs low.

• *Fact:* Costs certainly aren't the most accurate indicator of how good the software is . . . but you don't typically put retread tires on a brand-new Mercedes. Quality software — a system which has the quality you would expect help manage your organization effectively — is going to be a little higher priced.

There are lots of factors which go into software price — and probably only about 25% of them have anything to do with writing the programs (more on this in Part III).

• *Myth:* One great thing about a package is that we can install quickly . . . get it running in a month . . . maybe get several packages running in a month or so. . . .

• *Fact:* For one thing, remember that quality software firms are busy too . . . their schedules may not fit exactly with yours.

For another *YOU have to learn the package BEFORE your users do. You should run parallel (or at least develop a good test case). Your objective should NOT be just to see if the package works – but does it work for YOU.*

Take your time . . . do it right! After all, your organization will probably spend many months deciding on the right hardware — surely you can allow an extra month or two to make sure the software is operating properly.

• *Myth:* Our organization can't be TOO different from everyone else — we should be able to fit into a STANDARD General Ledger . . . STANDARD Accounts Receivable . . . a STANDARD Order Management System . . . a STANDARD . . .

• *Fact:* Data Processing should work for YOU . . . not the other around. Companies ARE different. Packages ARE different. Approaches ARE different. Features ARE different . . . and some are critical. A large multi-national firm doesn't just go changing its chart of accounts because THAT'S THE WAY THE SYSTEM WORKS!!!!!!!!

• *Myth:* Most applications are easy, straight-forward systems.

• *Fact:* Anyone who feels this way needs to design and implement from scratch just ONE Payroll system . . . just one INTEGRATED General Ledger to realize that there are NO easy applications.

If you find yourself saying to the software or hardware firms you are talking to, "We just need a standard A/P system . . ." catch yourself and re-think. You may be identifying yourself as the classic "easy mark." And easy marks have a way of giving their money and time to people who give little in return.

• *Myth:* The terms General Ledger, Accounts Receivable, Accounts Payable, Purchasing, Materials Handling, Resources Requirements Planning, Financial Modeling, and Inventory Control are universal . . . they mean the same thing to everyone.

• *Fact:* While your technical staff may think that General Ledge was probably just promoted from Colonel, ONLY your financial staff will be able to determine the features that are needed. Be careful . . . don't assume anything.

General Ledger does NOT necessarily include financial statements. Accounts Receivable systems don't always let you apply cash to ANY account (not just to

This paper is an excerpt from the book *A Success Plan . . . for Software/Implementation* by Steven J. Dennis and Barry Barnes. Published by Barry Barnes & Company, 1982.

an open receivable). Order Management differs radically for manufacturers vs. distributors.

Many systems do not allow for much FLEXIBILITY in your chart of accounts. Few packages allow you to customize the software to your organizations's own UNIQUE requirements (short of an almost complete rewrite of the software).

• *Myth:* A package is a system that is operating successfully at some other company . . .

• *Fact:* Wrong! Packages are WRITTEN to be PACKAGES!!! They are not — NOT — custom systems which happen to work at an organization which may be in the same industry as yours.

And, remember a package is also *not* something that is SCHEDULED to be done . . . it is something that IS done.

A TRUE software package is one *developed to be a package* by a firm that has *as its business* developing and supporting PACKAGES!

• *Myth:* All software packages obviously contain the proper audit trails and accountancy . . .

• *Fact:* Make sure your controller, chief financial officer, administrator, vice president of finance, business manager, or your CPA takes a good look at the package . . . we have heard of too many audits that insisted on changes to existing software . . . expensive changes . . . to include fundamental audit trails (the ones everybody *assumed* were there in the first place).

The people who write software don't always understand accountancy considerations . . . on the other hand, just because the firm has strong accounting credentials, doesn't mean that their software adheres. ASK about controls, audit trails, security, and the philosphical underpining of the software products.

• *Myth:* Just because you have a computer or are getting a computer . . . just because management feels that EVERYTHING should be automated . . . just because there are relatively inexpensive packages crying to be bought, we should surely bring ALL applications in-house.

• *Fact:* Some applications require careful analysis before a final decision is made to go in-house. Payroll is the best example: For small companies we have often recommended against the hassels of maintaining their own payroll . . . changes in tables, government forms, minimum wages, unions, etc., require a heavy in-house investment . . . well worth it for many organizations — but definitely not for everyone . . .

Be appropriate. Automate when there's some definable distinct advantage to the organization. Automate when you expect to be able to see *results.*

• *Myth:* For those of you whose organizations have an in-house data processing staff, "there just isn't ANYONE out there who can develop a system better than we can right here in our own organization" (also known affectionately as the "Not Invented Here" syndrome . . . or, often, more accurately, as the "Kiss of Death" or "Results Not Yet In" syndrome).

• *Fact:* First off, you are probably viewing it from the *technical* side . . . and from that viewpoint there may well be some truth. After all who better knows the DP philosophy, particular hardware configuration, internal politics, etc., better than your own data processing department.

In fact, the software house you select should be an expert in PARTICULAR applications — they know General Ledger, Order Management, Payroll, Medical Billing, Financial Modeling . . . and in the long run THAT's what you need.

And, by the way, (to the surprise of the DP staff) a good software package will often be impressive technically as well). This is much truer now than in the past. "Mature" packages are often relatively new — and often written using software development tools that simply weren't available a few years ago.

• *Myth:* With a wealth of new systems and languages, programming is now much easier than before . . . surely I can write my own applications.

• *Fact:* Programming is not so terribly difficult . . . but desig a particular function — or an entire system — requires the experts. The main benefit of the advent of powerful software development tools is that they free up time to do a more comprehensive job of DESIGN and that's fundamentally why packages are so suddenly such a viable alternative.

Learning to speak another language is one thing . . . writing a novel using this new language is quite another! And, writing that novel error-free? . . .

Programming is only approxiamtely one-sixth of the total effort ... the whole picture looks something like:

• Design	one-third
• Programming	one-sixth
• Test/Debug	one-third
• Training & Documentation	one-sixth

• *Myth:* We MUST have our programs written in COBOL, or some other such language.

• *Fact:* Arguing for a particular computer language is usually ridiculous. It's like arguing for French, Spanish or German — all of which are excellent languages.

Properly used — FORTRAN, RPG, COBOL, BASIC, and many other user-oriented high-level languages can provide *excellent* solutions.

Remember, even though *you* may think English or Danish, or whatever is the world's best language, millions speak others . . . write others . . . get results in others.

• *Myth:* Choice of language doesn't matter AT ALL . . . choice of file handling technique doesn't matter AT ALL . . . as long as it WORKS!!!!

• *Fact:* Buying something completely non-standard can be a disaster . . . insist on complete documentation for anything that looks a little out of the ordinary.

• *Myth:* Since my company deals with a single-person (or small) law firm . . . or a single-person or small CPA firm, it's okay to get my software from a very small software house . . . or even from an individual.

• *Fact:* This is a tough one . . . certainly there are many, many excellent, well-qualified software firms. Remember, though, our industry does not yet have a bar exam or any accepted professional certification like the CPA . . . nor are there ANY levels of standards throughout the industry which compare with the standard "generally accepted" accounting practices that all CPA's follow.

On the other hand, if your small (one, two . . . five-person firm) KNOWS their business — and yours — they *may well be far superior* to the 100-person company which views you as a small fish in a big pond. We know of a company — one of the 10 largest in the world — which actually PREFERS to work with small (one to ten-person) software firms.

Put this test to work . . . if my CPA went out of business, where would I be? Probably all right — there are others who could step right in. NOW if my software consultant/supplier went out of business then what???

• *Myth:* There are software firms to whom I can turn over complete responsibility for my implementation . . . total *turnkey* solutions . . . software houses which will sign a contract guaranteeing success . . . with penalty clauses . . .

• *Fact:* There ARE firms which will tell you that they'll take on all responsibilities. But let's face it . . . whose system is this? Theirs? Unless *you* take the responsibility for the solution to work . . . invest the time . . . invest the energy . . . adopt the right attitude . . . you will be developing all the ingredients for failure.

And, remember, desperate people will sign *anything*. A firm that knows how to do business doesn't have to sign your attorney's document in blood — they'll drop you like a hot potato and move on to someone else who knows how to get results.

• *Myth:* Custom software is never necessary or if it is, it should always be done in-house.

• *Fact:* Not true! Custom software IS quite often required.

For example: We have worked with a large client which fabricates and erects steel for many of the really *large*, modern high-rise office buildings and hotels in the Western U.S.A. Recently, a custom system was developed (by an outside firm) for this steel fabricator — one that estimates, to the nut and bolt level, the steel requirements for a 40-story office building . . . determines the best source throughout the world for that steel . . . how to transport it . . . and how long it'll take . . . and cost. That doesn't exactly lend itself to a package.

SUMMARY

There is an emerging — definite — context for standard application software packges. The degree to which workable solutions and procedures evolve for the successful incorporation of this new field into our business life directly affects the results we can expect for the near future.

The use of the data outlined in this presentation can assist in *initiating* the process of getting RESULTS . . . for your organization and for others.

Now, let's move on to an examination of the process of assessing software, selecting the software supplier, and implementing the software system.

PART II
SELECTING A SOFTWARE SUPPLIER

INTRODUCTION

The approach outlined in this section of this book is most appropriate for companies in the $10,000,000 to $10 Billion annual revenues categories. Smaller companies tend to be able to fit extremely well into totally "standard" packages — often being able to change their mode of operation to fit the package. Larger organizations — particularly when they hit to $25,000,000/year level — tend to have developed unique operating styles, unbendable procedures, inflexible managerial requirements, or just plain strong preferences.

In general, very small organizations can ignore a lot of what we propose in this book. Yet . . . the fundamental underpinning of RESPONSIBILITY for results — a commitment to being successful — will still serve well!

CONTEXT — A VARIETY OF VARIABLES

Packaged software has become the major area of focus in the information systems field. Yet, the industry called "Packaged Software Firms" has no equivalent of a FORTUNE 500. In fact, only recently has the situation arisen where there are more than a dozen companies in the world which have annualized sales of more than $10 million (and those few have only recently attained that level).

The vast majority of software firms — including the QUALITY ones — are small companies doing between one-half million and five or six million dollars per annum. And, small businesses are sometimes subject to radical ups and downs.

Thus, the selection of a software firm needs to be based on a set of criteria which optimize the potential for success. It may well be the case that the *firm* supplying the software is as important — or more so — than the software itself.

The important point to know . . . and acknowledge . . . and accept — whe you like it or not — is this: In selecting a software supplier, you are establishing a *long-term business relationship!* And, if you do your job WELL, you'll establish a really long-term relationship.

So DO your job well. Exercise the same care you'd use in selecting your CPA firm and your Corporate attorneys.

PURPOSE — A TOOL FOR MEASUREMENT

This set of guidelines encompasses what we've learned from our own experience as consultants and as software suppliers. It is developed from a variety of viewpoints. We offer it as a tool to use to measure any software firm which offers standard application software packages.

GUIDELINES — CHECKLISTS & PROCEDURES

Following are a series of "bullet-item" guidelines. These can be greatly expanded: These lists are by no means meant to be all-inclusive . . . they're simply a good start.

Give the software suppliers you deal with an opportunity to present their story to you — in their own way; then use these items as a checklist.

Don't expect any one firm to get an A+ on all items. We *know* that we're not "There" on them all . . . and we probably never will be! A score of 70 to 80% is "Excellent"; 90%, "Superior"; 100% . . . well . . . come on . . .

THE SELECTION COMMITTEE

Application software packages should NEVER be purchased by a single person . . . and that's not an indictment that *individuals* can't adequately make the decision. Some can.

The truth is: Software must be implemented by a variety of people at different levels in different functions within the organization. The wise organization will get a variety of viewpoints from the start.

In general, the Committee Model doesn't work in this world (as governments strive ernestly — and repeatedly — to prove). Yet, here is an example where a team of well-chosen effective people can make a real significant contribution.

Ideally, the Selection Committee should include representatives of the following functions . . .

- Organizational Management
- Functional Management
- User/Operator
- Data Processing/Technical
- System Implementor — the person who will have the responsibili of successfully implementing the system once it's chosen.
- System Coordinator.

A properly selected (and operating) Selection Committee can achieve tremendous results for the organization. It should meet frequently during the buying cycle. Each member should diligently fulfill assigned responsibilities.

THE SOFTWARE/SYSTEM ACQUISITION PROCESS OVERVIEW

We can't say what works for everybody . . . but in our experience as managers, users, consultants and software suppliers, we've found the following procedure to be a valid one:

Organizing Yourselves

- Define the Selection Committee
- Define the overall, broadbrush implementational schedule
- Develop a *brief* Selection Committee charge and guidelin
- Develop a *brief* background and requirements document fo the software suppliers

The Review Process

- Define the software packages to evaluate
- Poll your colleagues for additional ones
- Call your friends
- Ask around at social occasions and cocktail parties
- Poll the computer vendors for others
- Contact your CPA (and other consultants)
- Preview the various directories
- Define acceptable computer vendors
- Collect literature & documentation
- Contact the software suppliers
- Contact the hardware vendors

Our recommendation is an unusual one. We recommend that you do the front-end work yourself (previewing brochures, telephoning software suppliers, calling a few of their resources), and quickly narrow it down to three to five finalists which you'll then *visit* . . . and then send your RFP (if you use one) to only those. It saves you a lot of time and energy in the long run, while letting you be sure that the supplier who responds with a proposal actually knows something about your business.

The Preliminary Evaluation Process

- Develop a preliminary set of selection criteria for the software packages (Software Requirements Checklist)
- Develop a preliminary set of selection criteria for the software firm
- Find out more about the software firm (by telephone)
- Find out more about the application packages
- Procure the software supplier's client lists
- Telephone the software firm's references
- Select three to five (3-5) finalist firms
- Visit the finalist software firms
- Visit the computer vendors

The Interim Evaluation Process

- Analyze the findings of the visits
- Expand the Software Requirements Checklist
- Adopt the budget for software, hardware, etc.
- Develop the Request for Proposals (if appropriate)

The Final Evaluation Process

- Receive and review the proposals
- Prepare the Comparative Requirements Analysis
- Review the responses (entire Selection Committee)
- Review the responses with your users
- Review the responses with the software suppliers
- Select and advise the software supplier

The next step may be the most important part of the process. Make s you get to know ALL the people with whom you'll be working . . . and that they get to know — and like — all your key people. This is a people-oriented business. The better you know, and understand each other, the better will be the overall level of affinity and communication and reality when the going gets tough . . . and it *will*, at one time or another, get tough!

Establishing the Client/Software Firm Relationship

- Complete the financial requirements & procedures
- Complete the legal/contractual requirements
- Establish the technical support contact points
- Establish the software training procedures
- Establish the documentation update procedures
- Establish the user support procedures

In general, it's the hardware vendor who will maintain the computer equipment and the operating system. DON'T rely on the sales representative . . . he or she has other sales to bring in! Get to know the people who will support you.

Establishing the Hardware Vendor Relationship

- Establish the relationship with the hardware vendor's operating software and hardware maintenance staff (SE's & CE's)
- Complete the legal and financial requirements with the hardware vendor
- Issue a purchase order for the required computer hardware and operating software

Implementation Planning

- Define the Implementation Review Committee (may be the same as the Selection Committee)
- Adopt a requested Implementation schedule
- Present the requested schedule to the software supplier for review and resolution
- Review the software supplier's recommended Implementation Plan
- Mutually resolve discrepancies to achieve an Adopted Implementation Plan

General Application Preparation

- Schedule applications training
- Schedule technical training (if appropriate)
- Review procedures for potential modification
- Conduct weekly or bi-weekly Implementation Committee review sessions
- Procure the software supplier's final recommendation of the hardware configuration
- Finalize the hardware configuration

General Computer Hardware Preparation

- Conduct the site review for the computer
- Prepare the computer site as appropriate
- Analyze & define CRT and hardcopy printer locations
- Arrange for cabling and modem installation
- Conduct Implementation Committee review meetings

Final Implementation Preparations

- Initiate applications training
- Initiate hardware training
- Initiate other technical training
- Conduct project activities for special/custom work
- Conduct Implementation Committee review meeting

Implementation

- Install the computer
- Install the software module(s)
- Load/convert data to the new software
- Conduct application testing
- Conduct Implementation Committee review meeting
- Conduct end-user operational training
- Implement the application(s) on a "live" basis

On-going Review

- Conduct periodic review sessions with end-users
- Conduct periodic Implementation Committee review meetings
- Conduct periodic software firm review meetings
- Conduct periodic hardware vendor review meetings

This checklist is *one* way to acquire software. It's surely not the ONLY one . . . it just works! Try it . . . or modify it. But whatever you do, have a commitment to get results; develop a a Plan . . . then WORK the plan.

THE SOFTWARE REQUIREMENTS CHECKLIST (THE STATEMENT OF REQUIREMENTS)

The Software Requirements Checklist (also known as the Statement of Requirements) is the common thread for the *Functionality* of the software to be chosen. Most of what should go into this document is a list of *features* which you want or need . . . thus, it's impossible at this time to descri exactly what should be on it. Following, though, is a set of guidelines for preparing it:

- *Don't overlook the obvious* . . . don't "assume" that what want will automatically be in all packages.
- *Clearly state your requirements.* Avoid lots of text . . . it doesn't get read; use lists or checklists, where possible.
- *Rank your requirements.* Don't get too fancy . . . "A" for Must-haves; "B" or "3" for Highly-Desirable; "C" or "2" for Nice-to-Haves or Tie breakers.
- *Include philosophical or approach requirements* — things like on-line vs. batch . . . language . . . options . . . decentralized vs. centralized management style . . . growth . . . plans . . . security requirements . . . accounting batches . . . auditability . . .
- *Include interface or customization requirements*
- *Ask about product expansion* — are updates provided? How How often? What happens if you customize? Interface? Is a software support agreement available?
- *Ask about documentation* — what's provided? How readable is it? Who writes it? How often is it updated? Do you *get* it? How many levels of documentation are there?
- *Ask about training* — what's provided? How often? Who attends? Are there standard classes? Do you get the training materials? Who conducts the training? Who attends the training?
- *Ask about installation procedures* — what checklists wil you have? What procedures will be provided? What assistance will you get?
- *Find out about support* — what happens when you get in trouble? How do you report software bugs? What facilities exist for phone consultation? How often is user documentation updated? What procedures exist for follow-up on your requests? How do you suggest changes and the "wouldn't it be nice . . .," enhancements or extensions? Does the firm have a standard support program and a standard support contract?
- *Describe your organization* — tell the software firm abo your objectives (lists — not narrative); get each functional unit's requirements; get alignment so that you'll all be using the same terms . . . expecting the same results.

A well-designed Software Requirements Checklist should have terse one- and two-line features/requirements statements with a place to the right (or left) for the software supplier to enter a short yes/no/"*" response. Each *grouping* of features should be followed by a "*" blank space to write responses, exceptions, n rates, quotations for custom work, future release dates, etc. In other words, make it EASY to USE.

WHAT ABOUT CUSTOMIZATION?

No matter how much you may desire otherwise, your application may well have requirements that just aren't available from a standard package. We've seen requests for such applications as railcar tracking, event scheduling, loan tracking, event-driven action item management, and such — applications which are mainstream to the company, and which must reflect the *specific* approach of the organization.

The need for customization need not be a catastrophe . . . IF . . . you understand the implications. Learn how to define the need:

- Check against the Statement of Requirements for *alternative approaches.*
- Use the software firms' *consultative assistance* — they have suggestions as to how others have solved the same problem.
- Seek out *parameter-driven software* — it may be tailorab your need.
- *Re-evaluate* your need . . . if there's no package availa just may be that you're doing things too differently.
- Don't be afraid to *stick to your requirements* . . . perh organization's way — non-standard as it is — is the one which gets results!
- Determine whether standard package modifications required a *structural or are general enhancements* . . . that is . . . are you adding a room . . . or repainting . . . do you need a new foundation. IF the changes are structural — go custom (or buy the package to use as a building block for a custom system, with the understanding that you must take ownership of the resultant system).
- If you've found a "fit" or a near-fit for other applicati *discuss with the software firm* their interest in developing your custom requirements . . . or their recommendations . . . or how cooperatively they'd work with another firm which you might engage.

Most software requirements can be met from a package . . . but NOT ALL . . . not even all the "of course THAT would be in a standard package" requirements for a given function may be in the package you like best (and it may still be the best package to buy).

THE SOFTWARE DEMONSTRATION

Seeing the software work is extremely important. Software — like the people who design and use it — has a personality. Ask for a software demonstration!

The demonstration is best done at the software firm itself (assuming it has its own computer).

- Define in advance which modules you want to see.
- Send the software firm a copy of your Statement of Requirem in advance.
- Get the RIGHT people there.
- Come prepared.
- Give the software firm an extra hour or two to tell their story.
- Take the software firm's advice.
- Allow sufficient time.
- Ask for an extra hour or so — at the end of the session to unanswered questions handled.
- Tell what you expect to accomplish.
- Keep an Open Mind!
- Keep on Track.
- Be courteous.
- Ask Questions.
- Look at the Audit Trails.
- Look at the User Documentation.
- Look at the Technical Documentation.
- Look at the human engineering.
- Expect to have a reasonably good fit — (75% to 85% is considered a good fit; 90% is considered excellent; 100 is rare!).
- Expect to find gaps.
- Beware of the Everything's Great Syndrome.
- Be professional.
- Meet the user support staff.
- Don't demand to spend lots of time with the technical staff
- Follow up on Unanswered questions.

A good software demonstration can enlighten you as to omitted items on your Statement of Requirements. Also, it's an opportunity to interact with key people in the software firm who may end up being your contacts for years to come.

Use the demo effectively . . . then go back home and update your Statement of Requirements. *Have a Selection Committee meeting after each such visit.*

THE REQUEST FOR PROPOSALS

This is probably the area in which the *most* mistakes are made . . . by the requestor!

In the first place, use an RFP where it's *appropriate*. If you find a package during the preliminary search which meets your needs . . . offered by a firm that fits all the selection criteria, then, WHY go through the misery of an RFP? And, the RFP is as much work for you as for the software firm (if you're doing it properly).

Admittedly, it is a controversial stance to recommend *against* RFP's . . . but more TIME — and, often, MONEY — is spent by some prospective buyers, going through the drudgery (and motions) of Proposal Requests than is often spent on *acquiring and implementing* the software itself!

Let's look in more detail at the *purpose* of the Request for Proposals.

What an RFP Isn't

- It is NOT a way of justifying an already-made decision.
- It isn't a guarantee (those are up to YOU!)
- It isn't a fancy way of covering shoddy selection processes.
- It isn't a "test" for the software supplier (quality soft firms throw away the ones which look like "tests").
- It isn't a way of badgering others into doing things your way.
- It isn't a document more than ⅛" to ½" thick (anything thicker is a request for free consulting services).
- It is NOT something to be tied into a contract (most software firms which will agree to tie the RFP and their response to the contract aren't capable of living up to a court challenge; who *can* deliver what they say aren't interested in complicating their legal agreements . . . on strong counsel of their own attorneys!).
- It isn't, in summary, much more than a statement of requirements . . . than a statement of requirements . . . a Statement of Requirements.

What an RFP IS

- A Statement of Requirements.
- An opportunity for the software firm to tell its story (in *own* way).
- A place for summarization of costs.
- A place for summarization of potential implementation sched dates and events.

Guidelines for the RFP

- Make it *friendly!*
- Make sure you *personally* contact the firms you send it REPEAT: Personally telephone — or, better yet, VISIT — the software suppliers you want to respond.
- Send it to NO MORE than *three to five* firms.
- Tell about your organization (briefly).
- Give a brief background of the situation for which you're seeking solutions.
- Include the Statement of Requirements.

- Allow the software firm to answer in its own style.
- Attach an outline of what you want to know about the softwa firm. Keep it to the "need to know" level.
- Avoid rigid formatting requirements.
- Don't demand all sorts of contractual modifications . . . suppliers just aren't interested in doing business that way (no matter what counsel you get otherwise).
- Don't ask the supplier to tie their response to a contract good firms spend enough time on their contractual agreements to do business well — and their attorneys counsel them heavily against such modifications.

On the other hand, if the software firm has shabby-looking or weak contracts, you're entitled to ask for contractual modifications . . . but . . . do you really want to do business with such a firm?

- Remember: The software firm is busy — particularly if they're competent. If you ask for two weeks worth of work to respond to an RFP, you'll only get the software suppliers who aren't in demand.
- Get competent assistance in evaluating the responses.
- READ and ANALYZE the responses.

As a summation, regardless of whether you agree with us on the value of an RFP, don't let the RFP be a substitute for human interaction, client reference-checking, . . . software firm visits . . . and just plain hard work. If it's worth doing, it's worth spending some time doing right.

EVALUATING THE SOFTWARE FIRM

It is almost as — maybe even more — important to carefully assess the software supplier as the package itself.

REMEMBER: You are establishing a long-term business relationship. Just because this is a highly technical field, don't be bamboozled into anything. Look for the *same* organizational attributes that you'd look for in ANY company!

The successful software organization has to know their business . . . and must be committed to supporting clients using standard software products.

Following are some checklist items we've come across. Perhaps you can extract several items and develop a weighting/evaluating schema which will work for you.

- *Look at Outward Appearances:* How does the software firm va itself. What evidence is there that they've been around for a while . . . survived the magical "New Company Mortality" syndrome? How have they invested in the future? Look for "gut-level" feeling. DO they *look* like winners? Check these attributes:
  - Facilities
  - Equipment
  - Furnishings
  - Clients
  - Checklists & Procedures
  - Documentation
  - How the people view things (values)
  - Organizational Structure

- *Look at the People who Create the Packages:* Again, use gut-level approach. Would you want them working for your organization?
- *Look at the Firm's Background:* More gut-level stuff. Lo them and their experiences . . . and don't assume there's any one "right" way. Listen carefully to their story.
- *Look at the Knowledge Level of the People:* How have the all together? Does the firm have what it takes to succeed in ALL its areas (not just the technical)?
- *Look at their Guidelines and Standards:* Examine the evi of commitment. *Written* standards are statements of intention, stability, permanence. They're important!
- *Look at their orientation:* This is incredibily important . . . this is the firm's *real* purpose. Ask candid, yet open, friendly questions.
- *Look at the Firm's Target Marketplace(s)*
- *Check for Internal Procedures:* These are the indicators attention to detail. And, that's critical in the software field.
- *Look at the Support Apparatus:* This will be your contac AFTER the sale. You only interface with the sales or business development function BEFORE you sign up . . . IF you select the right firm.
- *Look at What Goes Into Product Price:* The difference in price is immense. There are, in many cases, equally-developed software packages available from different firms at vastly differing prices. What's the difference between a $3,000 Order Processing system and a $40,000, $50,000, or even a $100,000 one? Generally, there ARE features differences . . . but not always. The difference may well be in the firm. How does a firm selling their product for ten times that of another one of equal "features" survive? Generally, because they appeal to organizations which *value* long-term commitments.
- *Look at the Software Firm's Sales Style:* Here's where a of the true values of a firm show up . . . to the extremes. If the sales representatives act like a stereotyped salesperson, then there's probably something behind the scenes which supports such an approach. On the other hand, if the people responsible for business development show good knowledge, experience, and a consultative approach which demonstrates genuine concern for your success, they're probably reflecting some

very solid and fundamental philosophies and policies of the firm . . . latch onto them.

- *Look at the Implementation Planning Assistance:* The understanding of implementational considerations is one of the most positive indicators of a real-world, results-oriented firm.

  If the firm begins talking implementation, listen to them. Chances are, if they've passed the other tests, they know far more about how to implement a system than you. After all, they're probably doing it between ten and a couple of hundred times per year!
- *Look at the Legal/Contractual Instruments:* If you get a double-spaced typewritten page for a contract, feel free to take a hatchet and an army of attorneys to it.

  On the other hand, if you get a typeset, well-structured document that provides for mutual protections and which incorporates and documents business procedures, then expect the firm to be relatively resistant to modifying it.
- *Look at the Firm's Growth:* The software industry is in explosive environment.

  Unfortunately, even organizations with shoddy products and shabby outlooks can survive — even grow rampantly! Growth is a tough thing to handle . . . and it's roughest on the firm which is committed to quality.
- *Look at the Company's Management Style:* Despite all the growth, the firms which will *truly* succeed (for themselves and for you) are the ones which MANAGE themselves well . . . just as in *any* field, good management pays off.
- *Look at the Company's Business Ethics*
- *Look at the Software Firm's References:* Ask for references . . . AND . . . then *contact* them.
- *Talk to the Software Firm* — This has to be the most imp criteria of all. Talk to the software firm as you would to ANYBODY who could truly assist you; don't worry about giving too much of yourself and your values to them . . . if they're unethical, they'll *definitely* try to take advantage of you — be mature enough to be willing to find that out beforehand.

  Choose the software firm as you would your CPA firm or corporate attorneys. Choose them using the same criteria you would if your organization were going to acquire them . . . or if you were going to invest in them.

AFTER YOU SELECT THE PACKAGE . . . THEN WHAT?

There are several steps which should be taken. The important thing is NOT to assume that you're "there" . . . indeed, the journey is still somewhat in its infancy. In fact, you're just beginning! There's some more internal organizational analysis that needs to be done . . .

- Determine what people-problems you will have with software tha cuts across organizational lines . . . Order Processing for example can affect marketing, sales, credit, customer service, production control, manufacturing, shipping, quality assurance, AND accounting . . . what will your *people* problems be?

  Develop a plan for handling the inevitable . . . it WILL occur!
- If you have to modify procedures to fit a selected package, try it *manually* first. Get the resistance out of the way . . . PRIOR to having the computer to blame.
- Get the user to sign off on the system . . . the Accounts Re supervisor will be much happier if he or she blesses the system *in advance*.
- Take ownership of the system . . . and make sure everybody — including management — expects results . . . and is committed to doing whatever it takes to GET results. Finger-pointing and blame and "reasons" just simply have no place in the implementation phase. If they crop up, acknowledge them for what they are (the things people do when they're NOT getting results) and MOVE ON! (to getting results).
- If you haven't already done it, list your required enhancem Have the software vendor quote/ recommend how these enhancements should be done.
- Develop workarounds for all the functions which aren't exac the way you'd like the package to work — and inform everybody, so there won't be the excuse of: "Well, this package just isn't the way we should be doing things."
- Make sure you get completely trained on the software (from perspectives: User . . . technical . . . and standards). Make certain the user is fully trained . . . that there's the *ability* and *willingness* to understand.

- BE PREPARED — remember that users CAN damage themselves thr no fault of the software house or the software.

  Or even: "Advised of a schedule change??? Call the dispatcher."

  Convert some or all of the members of the Selection Committee into an *Implementation Committee*.

  Identify ONE person (for each module) as the System Implementor — that one person who has the ability and the responsibility for getting results . . . and who is recognized and respected by others as able and willing to make it happen. Have a meeting of the key players at least once every two weeks.

  Once the selection has been made, there's the cumbersome job of getting things rolling. And that's where the software firm's many experiences can assist you . . . that's where Implementation Planning and Project Control procedures come into play.

THE PURPOSES REVISITED

The purposes of this document are plain and simple: To provide at least *one* quality, honest, "what's so" step forward. Specifically, we offer a considerable amount of data, several methodologies or processes, and a sharing of experiences which may support you and your organization in attaining the successes you want. The only real value you can get from this book is a willingness to look at things *as they are* . . . followed by using whatever portions of our data, methods, and experiences which prove to be useful for you.

Data Communications Troubleshooting

Pete Fratus
Information Networks Division
Hewlett-Packard Company

PREFACE

Data communication problems can be extremely difficult to solve. They can also be solved very simply. Why the differences? Let's look at modern medicine for a few examples.

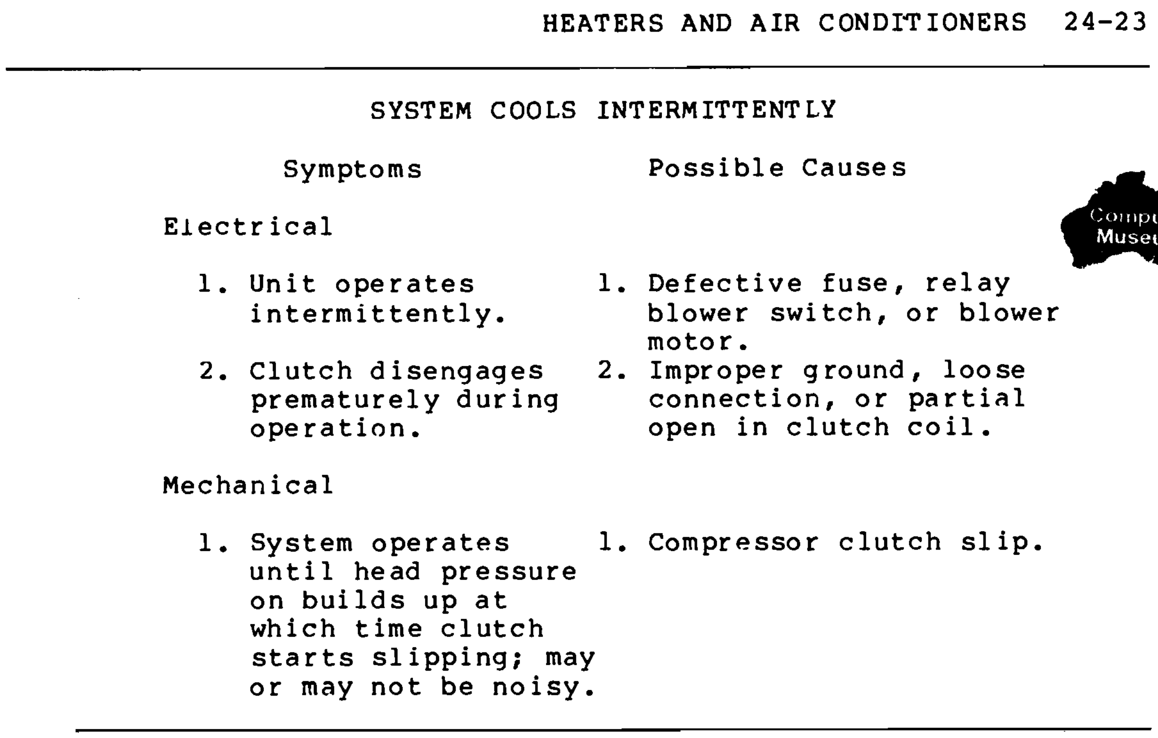
A patient complains of a sore arm. The doctor takes his temperature (they always take your temperature), examines his arm, asks some questions about past medical history and sends him to X-ray. Looking at the x-rays, she sees an obvious crack in the bone and places a cast on his arm. That was a fairly simple solution. Now take the same patient back to 18th century Europe. Tools for diagnosing broken arms were lacking, but there was always blood-letting. If that didn't work, the doctor could try irritants, Phrenology, magnetism and magic.

Had the doctor possessed the proper tools to do the job, the time between complaint and correct treatment would have been shortened considerably.

The type of problem, the tools available, and the technique applied can shorten or lengthen the time required to solve the problem. This presentation will help you understand the problem, become aware of the tools, and improve your techniques.

From the viewpoint of most computer users, there are four types of malfunctions. They are usage, protocol, digital and analog. Usage problems are those arising from improper use of an otherwise working data comm link. Protocol problems go beyond the users' immediate control and involve the software that handles the link. Digital problems involve the interface between the data terminal equipment (DTE) and the data communications equipment (DCE). Analog problems are limited to the data communications facility, which is the wires between the modems or data sets.

There are many approaches to troubleshooting. The process of elimination by replacing equipment, stepping through software, and circuit checks by the halving algorithm are some ways. Symptomatic troubleshooting does not eliminate any of these methods, but it does reduce the time necessary by quickly pointing out the area of the malfunction.



HEATERS AND AIR CONDITIONERS 24-23

SYSTEM COOLS INTERMITTENTLY

Symptoms	Possible Causes
Electrical	
1. Unit operates intermittently.	1. Defective fuse, relay blower switch, or blower motor.
2. Clutch disengages prematurely during operation.	2. Improper ground, loose connection, or partial open in clutch coil.
Mechanical	
1. System operates until head pressure on builds up at which time clutch starts slipping; may or may not be noisy.	1. Compressor clutch slip.

Example: Troubleshooting guide from auto repair manual.

An example of symptomatic troubleshooting can be found in nearly any automobile repair manual. You may find a flowchart or table in which the axes are labeled SYMPTOM and PROBABLE CAUSE. The idea is to find the probable causes for the condition (or problem) encountered, then by testing or a process of elimination, discover the remedy. Newer methods have been developed which can suggest solutions.

AUTOMATIC TRANSMISSION 21-30

General Diagnosis Chart

Symptoms	Causes (see list below)								
	1	2	3	4	5	6	7	8	9
HARSH ENGAGEMENT FROM NEUTRAL TO D OR R	X			X					
DELAYED ENGAGEMENT FROM NEUTRAL TO D OR R		X		X	X	X	X	X	X
RUNAWAY UPSHIFT		X		X		X		X	
NO UPSHIFT		X		X		X	X		
3-2 KICKDOWN RUNAWAY		X		X		X			
NO KICKDOWN OR NORMAL DOWNSHIFT				X					
SHIFTS ERRATIC		X		X		X	X	X	X
SLIPS IN FORWARD DRIVE POSITIONS		X		X		X	X	X	X

1	Engine idle speed too high
2	Hydraulic pressures too low
3	Low-reverse band out of adjustment
4	Valve body malfunction or leakage
5	Low-reverse servo, band or linkage malfunction
6	Low fluid level
7	Incorrect gearshift control linkage adjustment
8	Oil filter clogged
9	Faulty oil pump

Example: Symptom/Cause Chart from auto repair manual

Let's get back to the computer. Think of all of the possible problems one can have with a data communications network: hangs, disconnects, errors in the data, delays, retries, and on and on. What could be the possible causes of these problems? Noise, fade, delay, program bugs, faulty equipment, operator error, and more can all cause aggravating malfunctions.

This is what you need to know to solve these problems in a timely manner:

A. The Basics — what is the environment, what was supposed to happen
B. The Symptoms — what did happen
C. The Causes — why
D. The Tests — what tests will give the right information
E. The Tools — what tools will do the tests
F. The Solution — what action to take

This presentation will help you learn how to get from A to F.

EXCERPT

Let me give you an example to show how knowledge of the symptoms, tests, and tools can make problem solving easier. A problem occurred at a site where the symptoms were terminal hangs and garbage on the screen sometime after the session started. There was never any problem signing on. This was a point-to-point terminal on a switched line using BELL 212A modems.

Three things were done in an attempt to resolve the problem: the MPE I/O configuration was checked, the modem options were verified, and a 1640 data scope was put into the line. The 1640 showed that a DC1 was received followed by garbage. This was either printed as garbage or hung the terminal. The assumption that followed was that something was wrong with the terminal. The configuration of the terminal showed that it was "providing clock," as was the 212 modem. At this point, it was decided that the modem options listed in the Data Comm Handbook must be wrong.

This bit of troubleshooting had gone way off on a wild goose chase. No attempt was ever made to test the most basic part of the network, the telephone line. Simple modem self-tests and loop backs were completely ignored. The 1640 served no useful purpose at this point.

A more reliable approach would have been to start by defining the exact symptoms, determining the possible causes, and making some appropriate tests. Using the new information gained through this technique, troubleshooting would have been more directly related to the problem.

The Basics
HP262X terminal
Point-to-point terminal connection to a port
Switched public line
Full duplex modem with good complement of diagnostics

The Symptoms
Apparently random terminal hangs and garbage occurs only on one line

The Causes
Fortuitous line problems
Faulty modem
Faulty terminal

The Tests
Modem loop back with test pattern
Modem self-test
Terminal data comm test

The Tools
None needed

The Solution
Switched lines are susceptible to noise and other problems. Since each new connection uses a different route, conditioning is not available (and would not help noise anyway). Therefore, the terminal should be reset if it is hung or the data retransmitted if it was garbaged. If that doesn't help, redial the connection.

SUMMARY

This excerpt is an example of what my presentation will cover in much more detail. I am currently working on flowcharts and decision tables to make solving data comm problems easier by encouraging the use of symptomatic trouble shooting. This should lead to using the proper tools in the proper order.

Financing Quality Solutions

Melissa J. Collins

Is your manager a fire-breathing dragon? Does your budget get thrown in the dungeon year after year? If you answered "yes" or even thought twice about one of these questions, you are not alone.

The Data Processing manager faces many challenges and pitfalls in operating his or her department. One such pitfall is the budgeting and finance area. This paper will discuss solutions and methods to deal with department monies (or lack thereof) and interaction with a non-technical manager.

THE SUCCESSFUL DATA PROCESSING DEPARTMENT EQUATION

Good Equipment + Good People + Money = Quality Solutions

All of you have made an excellent choice in equipment. If you don't have the good people, they are out there for the hiring. Now all you need is the money and the management's support and Quality Solutions will be within your grasp.

So where does the DP manager fit into the money part of the equation? The manager submits a budget of his monetary wants/needs for a fixed period of time. Of course, just because he asked for a million dollars, doesn't mean he gets that amount. The DP manager must convince the upper management that the monies requested are sound investments in the company's goals and futures. This is where the hard part comes into play. HOW DO YOU convince the upper management, who has little or no computer training, that your budget goals are not unrealistic?

GETTING YOUR BUDGET APPROVED

First of all, you must face three facts. Once you come to grips with them, the outlook will not be so gloomy.

1. Your management is not against you or your department.
2. Getting the monies necessary to finance any DP project is sometimes harder than bleeding a rock.
3. Anything truly worthwhile is worth a small battle.

The first step to insure approval of your budget is to be realistic about your requests. But, at the same time, do not under-budget your department. This is a fine line to walk, but it can be done. It is always nicer to come in under budget than over budget, but if a department is consistently under budget, then a manager can be accused of "padding the budget." A few guidelines to consider about budgeting items other than normal expenditures (salaries, maintenance contracts, consumables, etc.):

1. If there is the slightest chance that you will need a new piece of equipment, budget for it.
2. Be sure you have sufficient justification for new equipment.
3. If using the budget as a tool for justification of new employees, provide good evidence of need. (Such as project time tables, department workload, etc.)
4. As a tradeoff — instead of new employees, budget for programmer productivity tools whenever possible. Offer this as an alternative to your manager. If the same results are achieved, the lower cost alternative will always be chosen.

If a manager is dealing with a non-DP superior, having his support is very helpful. The time you spend educating a non-DP manager is time well spent. If your manager knows what a disc is and what its uses are, getting approval for a new one is not quite as painful. The Data Processing department is surrounded by an aura that threatens some managers. The high technology and computer "buzz words" are enough to scare off anyone who doesn't know what is going on. By working with your manager and educating him, you will find that he will support you more. The old adage, "You can lead a horse to water, but you can't make him drink," applies here. Some managers could care less about learning more about the DP department. Subtle tactics can be used to educate a non-technical manager. Such tactics include, but are not limited to, inviting your manager to participate in your weekly staff meetings, taking him on a tour of your facility BEFORE you present your budget requests, or taking him to a regional user group meeting. These actions may prick his interest to learn more about the DP department.

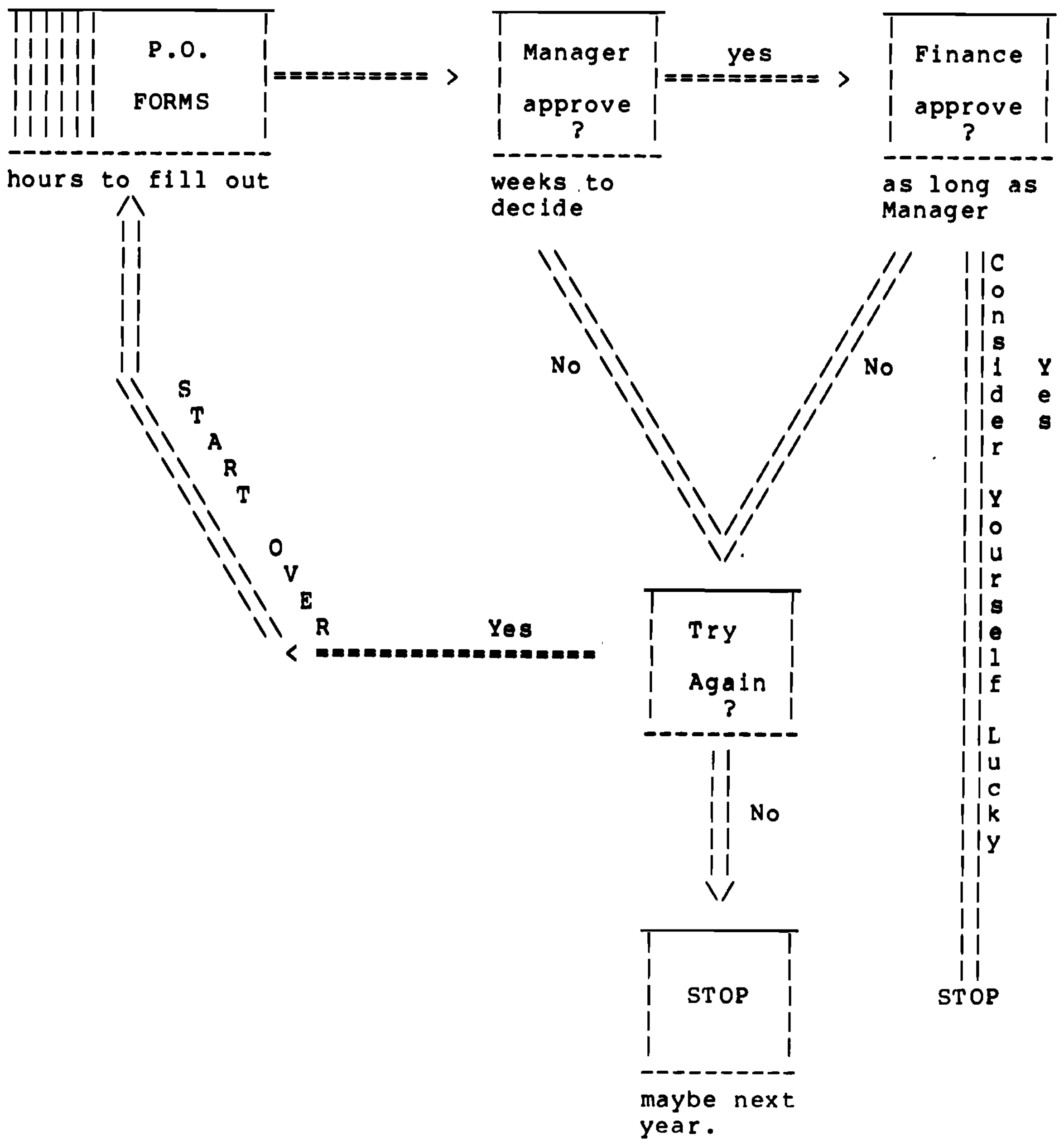
Interaction with users may not seem really important in attaining your financial goals. But consider this; if the users are unhappy with the DP department, this attitude will filter up to the managers of said users. The managers will, in turn, convey this attitude to the higher management who may ultimately be in the position to approve or disapprove your budget. You can't expect your users to be happy all of the time, but shoot for making them happy as much as you can. It will help around budget time.

USING YOUR BUDGET FUNDS

Now that your budget is approved, you don't have a free reign in distributing the monies awarded to your department. This is a fact everyone has to face. This problem has a name well known to all of us — RED TAPE. The government doesn't hold a monopoly on the man-made phenomenon. But, there are ways to circumvent its nasty powers. Take purchase orders for instance.

Instead of filling out two tons of paperwork for a purchase order number, have the vendor submit a bill to you that you can sign off. The end result is the same; you get product/services and the vendor gets the money (probably in shorter time). If you can follow the chart below, you can trace the path of the purchase order request. If a bill was sent to you, you could eliminate all of these steps.

PURCHASE ORDER PROCESS



Most, if not all, managers have a monetary limit for which they are allowed to sign without having their managers approval. Using this limit will help you eliminate not only the purchase order request cycle, but will also help you acquire products/services without the approval cycle. If you want a vendor's product that is $7,500, but don't have said product in the budget for this year, don't give up hope. Use the art of finagle. First and foremost, check your budget very carefully to see if you have any extra money anywhere. If there is some money, but not enough, consider cutting something out, like a tape cabinet. Then, working with the vendor, try to agree on some financial arrangement that will get you your product, the vendor his money, and, most important, not get you fired. Most vendors will work with you on this. Instead of not making a sale, they will gladly bill you on an installment plan. As long as these installments are under the amount you are allowed to sign for, you will all be getting what you want.

However, it is not wise management to practice the art of finagle all of the time. The more you use it, the more likely you are to get in trouble. It is to be used when the political climate of your company is not conductive to spending, or when you know that there is no other way to get what you want. Dealing "under the table," so to speak, is a tool you have available to you as long as you do not abuse it.

FIGHTING BATTLES

So what if it's in your budget? Your manager could change his mind by the time the purchase order hits his desk. What recourse do you have except retreat? You can stand up and fight for what you want, diplomatically of course.

> Di-plo-ma-cy (di plo'me se) n. 1. the conducting of relations between nations 2. the skill of doing this 3. skill in dealing with people; tact SYN. see TACT

According to Webster, the art of diplomacy is based on the skill of dealing with people. For a manager to achieve his goals, he must know how to deal with people. When a proposal is rejected, it can be very difficult for a manager to understand why his request was turned down. The first reaction is usually one of anger or frustration, which if vented on your manager, will leave little or no chance for a reverse decision.

When you have an important request turned down, analyze the situation. What was the reasoning behind the decision? The political climate of your company may have wreaked havoc on your proposal. Or you did not justify it in a way that your manager could appreciate your true need for your request. Try to see the situation from your manager's point of view. Discuss it with him, and ask him why the request was turned down. Try not to put him on the defensive about his actions. If any of his doubts can be resolved, do so as soon as possible while the situation is fresh in his mind. If this fails, you still have ways in your grasp to continue the fight.

The battleground is already set. You want something you feel you need, and your manager told you that you cannot have it. Whatever his reasoning was for denying your request, if you still feel very strongly that you need this request, sound the charge.

For an example, you have requested $3750.00 for a new memory board and your manager says no. Examine your reasons for this request. Obviously you feel the machine is slow, and your users are starting to recognize this fact. With your continued development work, the situation will only grow worse. Direct a memo to your manager explaining the situation as clearly as possible. Include in your comments that the users are starting to show dissatisfaction with the response time of the machine and that the situation will get worse. Other points to consider for mention are a slump in productivity and decreased throughput. If your General Ledger is coming up on end of month close, illustrate the consequences of productivity slumps and decreased throughput. That will make him stand up and take notice.

The memo serves two purposes: one, it informs your manager of the consequences of the denied request, and, two, should the situation not resolve itself, when the complaints start pouring in, you have documented proof that you tried to rectify the situation.

When the users do start to complain, and they will, tell them what the problem is and what you have tried to do about it. Be sure that you do not, under any circumstances, criticize your manager in this discussion. Eventually, the user attitude will be translated up the line to other managers and the pressure will be shifted onto your manager. This is the long way to go around the mountain, but you will eventually get what you want/need without stepping on anyone's toes.

Should this approach fail, give your manager alternatives. Instead of getting another memory board, consider optimizing your machine. In presenting this concept to your manager, provide as many solutions as possible. You could hire another programmer to optimize all of your code. You could buy OPT/3000* and train one of your staff to interpret its data and optimize where necessary. Or you could hire a consultant and let him figure it out. All of these are alternatives. When presented to your manager, he will realize that these are all costly alternatives, much more than a memory board. So in order to save the company money, you will most probably get your original request.

If all other efforts fail, start a memo blitz. Every week or so, send your manager yet another memo concerning the subject. Be sure that these memos are inoffensively worded or the heat will turn against you. You may get your request just because your manager wants the paper barage to stop or he doesn't want to be bothered anymore.

One final word on the subject. If you are informed that the answer is still NO after all of your efforts, it may be best to hold off for a while until the smoke has settled and then try, try again.

*Not to be considered an endorsement of this product.

Management: Key to Successful Systems Implementation

Gary A. Langenwalter
Manager, MIS
Faultless Division
Bliss & Laughlin Industries
Evansville, Indiana

When I arrived at Faultless four years ago, the new on-line Order Entry system was supposed to be completely operational. I found a completed general design, some detail design, and 10 programs coded. The hardware vendor (not HP) had promised Faultless management that they would contribute one person for one year, we would do likewise, and the result would be a state-of-the-art order entry system. We finished 1½ years late, with an investment of 6+ years of effort. We are currently replacing our old hardware with an HP3000, and replacing all our software. This conversion was scheduled to take 14 months, finishing November 30, 1981. Our best current projection is August 1982. In all fairness, I must mention that the Master Scheduling package that we bought three years ago was installed on time, under budget, and it met our expectations.

We at Faultless are not alone. Consider the following three disasters, all of which occurred in Fortune 500 companies in 1980:

> "A major industrial products company discovers one and a half months before the installation date for a computer system that a $15 million effort to convert from one manufacturer to another is in trouble, and installation must be delayed a year. Eighteen months later, the changeover has still not taken place.
>
> "A large consumer products company budgets $250,000 for a new computer-based personnel information system to be ready in nine months. Two years later, $2.5 million has been spent, and an estimated $3.6 million more is needed to complete the job. The company has to stop the project.
>
> "A sizable financial institution slips $1.5 million over budget and 12 months behind on the development of programs for a new financial systems package, vital for day-to-day functioning of one of its major operating groups. Once the system is finally installed, average transaction response times are much longer than expected." (McFarlan, p. 142)

Ollie Wight, the leading consultant in the manufacturing systems field, estimates that there are fewer than 25 "Class A" MRP users in the country! That number compares poorly to the multiple thousands of companies that have tried to implement manufacturing systems, each with the intent to succeed. We are one of the "thousands"; we are working to become "Class A."

The major risks of systems implementation can be categorized as follows:

1. Failure to obtain all, or even any, of the anticipated benefits.
2. Costs of implementation that vastly exceed planned levels.
3. Time for implementation that is much greater than expected.
4. Technical performance of the resulting systems that turns out to be significantly below expectation.
5. Incompatibility of the system with the selected hardware and software. (McFarlan, p. 143)

Three factors that determine the degree of risk are listed below:

1. Project size. The larger the size, the greater the risk. Size is also relative — a $500,000 project has much greater risk for a $20,000,000 company with a 3 person MIS staff that has never installed anything of its size, than for a $200,000,000 company with 20 programmer/analysts.
2. Experience with the technology. Unfamiliarity with the computer in question, or its operating system, or database, or TP monitor and terminals increases the risk.
3. Project structure. Having clearly defined inputs and outputs, which all users agree upon beforehand substantially reduces the project risk. I have not yet seen this, but it is theoretically possible. Conversely, when people are still changing basic systems functions and designs midway through the project, that project is doomed to overrun both temporal and financial budgets. The military is particularly adept at this (aided and abetted by the contracters).

My current experience, plus my previous background as an educator and consultant with a major DP hardware vendor, support the hypothesis that forms the basis for this talk:

HYPOTHESIS

The single factor most responsible for success or failure of system implementation is management. Good management requires identification and minimization of risk of failure, plus continual execution of the three basic management principles: Planning, Organizing, and Controlling.

The implementation of a system will be successful if, and only if, it meets three basic goals which are the converse of the risks listed above:

1. On-time completion.
2. On-budget completion.
3. The completed and installed system must meet both its specifications, and the users' expectations.

Let us review in some detail how each of the basic management techniques can be used to insure successful systems implementation.

PLANNING

Perhaps the best way to approach the topic of planning is with a cursory overview of the techniques available. Both PERT charts (or CPM charts, or "Bubble charts") and bar charts have been widely used for years. Appendix B includes a sample of each. In general, computer programs are a tremendous help in handling complex PERT charts, and recalculating critical paths.

Time estimating is perhaps the biggest stumbling block to proper systems implementation time and cost projections. Various articles suggest that each person on a project be scheduled at only 70% efficiency, and that one should allow 2-3 weeks for a user decision. My own personal experience indicates that one should allow 1-2 months for vendor feedback (to an RFP, for example), and for scheduling vendor presentations and reference visits. Also, if a person is managing others, 20% of his time should be allotted for each person managed, subject to the discretion of the estimator. Finally, an estimator needs to allow "Contingency time" of 20-200%, depending on the tightness of the other estimates, and the degree of risk inherent in the project — the contingency factor should increase proportionally with the risk.

Now, down to the actual planning itself. In my opinion, the only intelligent way to implement a large system is to break it into four phases, with management, the users, and MIS mutually agreeing to the functions, cost, benefits, and time estimates at the end of each phase. This minimizes the risks involved, and maximizes the probability that the user department will implement the finished product successfully. We will examine each phase below.

1. Initiation Phase

This phase includes the preliminary survey, a rough estimate of potential costs and benefits, and the selection of the alternative perceived to be the most attractive (make vs. buy, Vendor A vs. Vendor B, etc.). It culminates in a presentation to top management of the Systems Proposal written jointly by the user department manager and the MIS Manager. If top management approves, the system implementation enters phase 2. If not, it can be reworked or dropped, with minimal expenditures of resources to date.

This phase is the most important of all; it creates the basic expectations of system functionality in the minds of users and management. It should be noted that the basic system functions are defined by the person who will use them in his daily work, not by the MIS department representative. "Systems are tools for the manager, not toys for the technician." (Wight, p. vii)

Some of the topics which are covered in the Systems Proposal (or Management Overview) are management summary, major system benefits, economic justification, and schedule. Appendix A1 contains a more comprehensive list of topics included in the Systems Proposal.

One other topic which needs consideration throughout the implementation of a new system is the fear of change of the part of some people in the company. Some will be afraid that they will lose their jobs; others that they will not be able to measure up to the new expectations; and still others that they will lose their status with their peer groups, and/or that their work groups will be reorganized. These fears, unless addressed, can result in passive or active resistance to the new system on the part of the people whose daily enthusiastic cooperation is an absolute requirement. They must, therefore, be actively addressed and overcome.

2. Analysis Phase

This phase starts with a study which examines in greater detail all major assumptions and promises of the original proposal. Greater attention must be paid to any area that includes major uncertainty (response times with the particular hardware, application, and database under consideration, for example). Cost and benefits estimates are updated with the new information. My experience indicates that costs almost invariably increase, and benefits almost equally invariably decrease. Finally, the MIS department writes the Functional Specifications for the proposed system, and has them approved by the user department(s) affected. After they all agree, they jointly present them to the Steering Committee, with updated costs and benefits. If management approves, the project continues; if not, it is either discontinued, or revised. At this stage still, there has not been a major expenditure of corporate resources.

The Functional Specifications (or General Design) document can include the major logic chart, proposed input and output layouts, a training plan, future capabilities, and a contingency plan, to name but a few of the many topics. A more complete list appears as Appendix A2.

3. Design Phase

This phase defines how the system will be built. It is finished upon the completion of two major documents: the Design Specifications (or Detail Design), and the User's Manual.

Some topics that the Detail Design Specifications include are a detailed system flowchart, with input and output defined for each program, security, detailed program functions, specifications, and logic, and a detailed project implementation schedule. Appendix A3 contains a more exhaustive list.

One item that must be covered in appropriate detail is the Contingency Plan. All hardware, even HP's, and all software, even Faultless', will eventually fail. Such an event cannot be allowed to totally stop a critical department from functioning.

The User's Manual includes pictures and descriptions of all input and output screens, and reports, with explanations of all fields — what they mean, and how to change their contents, if appropriate. It also includes operating instructions (how to sign on to the system, what to do in case of problems, etc.). It must be written in language that the person in the functional department will easily understand.

These two major documents, plus Contingency Plan, are jointly presented by the user department manager and the MIS manager to the Steering Committee, with the re-revised cost/benefits data. If management approves, the system enters the final phase of implementation. If not, the minimum resources possible have been expended thus far; the project can be either revised or dropped. At this stage, all parties involved will have agreed on the details of the new system; there should be no "surprises" from here on out. There should be no reservations about technical capabilities, or about what the system will do.

4. Construction Phase

This phase is the one that includes the actual programming, testing, and documentation. In a well-managed project, more than 50% of the time should already have been spent designing. This minimizes changes, revisions, etc. that are the bane of efficient and effective systems. Let us discuss each subphase independently.

Programming is a complex enough topic that it warrants books, talks at this convention, and week-long training courses. Let me outline my views briefly, and then continue with the subject at hand. All programming should be top-down, structured, and modular. Each program or module must be tested and documented as soon as it is completed. It is then, and only then, that it can be included in the account that contains completed programs.

I will knowingly raise a controversy by suggesting that users should design their own screens (with V/3000, where applicable), and write their own reports (we are using REX for that purpose). To me, the data belongs to the user. Assuming that he understands the contents and implications of the numbers that exist in the database (and he should, for in most cases we hold him responsible for their accuracy), then he should be given the tools to generate the reports and inquiries that will allow him to manage his portion of corporate resources optimally. In other words, I refuse to perpetrate an "IBM" (International Brotherhood of Magicians) image with regard to my department.

The Systems Manual is a major document. It needs to follow predetermined specifications and formats, and, more importantly, must be updated throughout the life cycle of the system. There are very few things more dangerous than a slightly outdated Systems Manual in the hands of a programmer who is trying to maintain a system.

The Operations Manual is a must, whether your MIS department has a formal operations group or not. This document tells the operator how to run the batch jobs, back up the system, recover in the event of failure, where to send the output, etc. It defines expectations. If there is no formal document, the person who normally runs the job is generally the only person in the company with that information. The financial risk that represents to a company increases with the importance of the application (for example, weekly payroll).

Training cannot be overemphasized. The responsibility for training users lies with the Project Manager (the user department manager) rather than with MIS, because the head trainer becomes the person who knows the application better than anyone else in the organization. In smaller organizations, the Project Manager will train users directly; in large organizations, he will train other managers, who will then train their own people.

Training can and must commence as soon as the first few programs are finished. After the Project Team has trained itself, it is time to start familiarizing other personnel with the screens and reports they will be using soon. These people can often suggest invaluable improvements, some of which take almost no time to incorporate. The ones that involve much time must be prioritized, and approved by the Steering Committee prior to inclusion. The end users will also spot program flaws that escaped everybody else.

All user training and all program testing, except volume and response time testing, must be performed on a small test database, preferably one distilled from your real live database. My user personnel respond much more favorably to reports which include casters that they do to reports which feature bicycles.

Training is the one place that most people grossly underestimate the time and resources required for a proper implementation. Most people also underestimate the numbers of people that must be trained, and perhaps even educated. Training materials can be acquired from the vendor, if the software is purchased, and from video-tape training companies such as ASI and Deltak.

One has three choices for final testing: Parallel testing (which works well for financial systems, for example), Pilot testing (which can be used for some manufacturing systems implementations), and None (which I cannot recommend; the only cold turkey that I like is that which is left over from Thanksgiving dinner).

Final testing also quickly unearths any latent run time or response time problems. Although painful, and embarrassing, it is better to discover those problems at this stage than to try to squeeze 25 hours of processing into a 24 hour day after the old system has been cut off!

After the final testing is complete, one faces the actual conversion. Although this sounds simple ("Just take the old data and load it into the new database."), it can be most complex. Each type of data to be converted must be examined. Each outstanding piece of paper must be considered (Do we leave it there? Replace it? How do we find them all? What about the ones we miss?). To illustrate the complexity of such a task, consider that it took us at Faultless the entire Labor Day weekend, running around the clock, to cut our MRP database over from our other (non-HP) computer to the Series III. The process involved over 30 steps. The process and programs were so complex that we ran test runs on the conversion programs themselves several times.

ORGANIZING

Since the most important person in an implementation effort is the Project Manager, let us start by briefly defining his (her) attributes and responsibilities.

The Project Manager, in my opinion, must come from the department most affected by the project (that is, the one that will gain the most if it succeeds, and lose the most if it does not). It should be the person who will manage that function on a day-to-day basis after the system is successfully installed. The MIS Manager should be Assistant Project Manager, to insure that what the user wants is technically feasible. On a major project, the Project Manager position involves a full-time effort, especially when training commences. I know that in the "Real World," those people are often totally busy just keeping the company running on a day-to-day basis. But nobody else has the intimate knowledge of how that department really functions on a daily basis that is required for successful design and implementation of the new system. Faultless top management backs this philosophy 100%, by saying that if a department is not interested enough to furnish a Project Manager, the project will not commence.

The Project Manager is responsible for writing the functional specifications at the commencement of the project. They form the basis for all subsequent development. In my opinion, if a company does not have the time to write its own Functional Specification, (or RFP), and feels that it must hire a consulting firm to give birth to a 250-page document, that company has no business trying to implement any system that arises from that document, because it will not be "their" system. That system stands, in my opinion, a better than 90% chance of failure.

The Project Manager must plan, organize, and control (in other words, Manage) the day-to-day efforts of the project. He must continually check to make sure that detailed designs will meet the needs of his (and others') departments. He must monitor progress to schedule, and adjust the schedule to the realities that intrude on the best plans. He must control requests for changes by sitting on most of them, and presenting the few worthy ones to the Steering Committee. He must chair the Project team at its weekly meetings, and the Steering Committee at its bi-weekly meetings. As mentioned earlier, the Project Manager is also the head trainer, and trains either user personnel directly, or their managers who in turn train their subordinates).

The Project Team is comprised of the Project Manager (Chairman), MIS Manager (Assistant Chairman), managers of all departments affected, and the analysts and programmers assigned to the project. It is responsible for resolving differences of opinion that do not involve policy or fundamental operating philosophies, recommending policy and operations changes to the Steering Committee, ensuring that the project progresses as scheduled and results in the benefits promised, and prioritizing the myriad requests for changes, modifications, enhancements, etc. that occur in such projects. It must also monitor the creation and installation of internal controls, and contingency plans.

To be effective, the Project Team needs to meet weekly (a standing meeting time and place is usually appropriate). They need to keep a formal "Problem List," with the status of each problem, including its final resolution and date. This will ensure that a problem does not get ignored until it becomes extremely costly to resolve. The Project Team must send minutes of its meetings to the Steering Committee, with the Outstanding Problem sheet attached, annotated to show how each problem will be resolved. Finally, the members of the Project Team must be the ones who train on the new system first, and best. They will be assisting their subordinates to use the system correctly; they need to understand well how it works. They also need to know the inner workings of the new system so that the many decisions that must be made during an implementation will be the best possible.

The Steering Committee is comprised of the Project Manager (Chairman), MIS Manager (Assistant Chairman), the top executive of each department affected ("mahogany row," if you will), and the person to whom those executives report (the "corner office"). This committee should meet bi-weekly (more frequently during a "crunch"), to monitor progress, set policy, commit resources as needed, resolve any differences of opinion that could not be resolved by the Project Team, and approve/disapprove Project Team recommendations. It should not get involved in the day-to-day implementation effort; that is why the Project Team exists. The Steering Committee must also ensure that adequate contingency plans, internal controls, and documentation exist as the system is being designed and installed.

CONTROLLING

There can be no control without adequate plans, for one must control to a predefined goal. There can be no controls without proper organization, for there would be no person held responsible. Given, however, that plans and organization exist, control is absolutely mandatory. Without control, there is no feedback to inform management of deviations from plan to allow them to redirect the implementation efforts appropriately, or to measure the performance of the persons involved. Of the three management functions, controlling is the most difficult, and the one that is least well executed, in my experience. More implementation efforts fail from lack of adequate control than from the other two functions combined.

We have discussed earlier that the Project Manager, and Project Team, must control the project on a daily basis. They must monitor progress against each of the major requirements:

1. *Time*. To do this, each project must be subdivided into tasks so small that each of them takes one person no longer than two weeks. Each of these tasks needs to be identified on a PERT chart, staffed, and tracked. This avoids the surprise of learning, one week before scheduled conversion, that the project is six months late. Progress must be reviewed weekly.
2. *Budget*. The easiest way to monitor this is to use project control software. Expenditures must be reviewed weekly, in concert with progress and projected completion dates.
3. *Benefits*. These need to be followed also, for if they are not going to be achieved, the project should be considered for immediate discontinuation by the Project Team and Steering Committee.
4. *System Performance*. Same as Benefits. If the system will not perform as expected, implementation should be stopped unless reapproved by the Steering Committee.
5. Internal Controls, and Contingency Plans. In the euphoria of system development, nobody wants to think about such things. They are absolutely essential. Internal controls can, and do, highlight system deficiencies. After our new Order Entry system had been installed for a year, our Controller insisted on installing another simple internal control. It revealed that on a very few occasions, we were not invoicing our customers for goods shipped! Contingency plans are required, because the hardware will eventually fail. (Murphy was correct; ours failed during our monthly close.) We are still in business because we had developed contingency plans.

Let me reemphasize that the Project Manager and the Project Team need to continually keep the project boundaries in firm focus. I suspect that more projects have floundered and finally sunk from the mid-stream addition of features, enhancements, etc. than from any other single cause. Once the Functional Specification is approved, there should be no major changes without Steering Committee approval. Once the Detail Design is finalized, there should be few if any changes allowed.

If a package is being installed, requests for change should be segregated into three categories: a) Must Have Before Implementation, b) Should Have As Soon As Possible, and c) Nice To Have. There should be very, very few changes in category a); these are the ONLY changes that should be permitted before implementation of the standard package. Once the package is installed and running, over half the requests in categories b) and c) will disappear; they will no longer be necessary. Each change that is permitted to delay the installation of the package delays the benefits that will be derived from installation, and increases future maintenance problems.

The Steering Committee must measure progress against plan for all major dimensions outlined above, and ensure timely completion to specification. They must resist the overwhelming urge to modify, or enhance, unless the benefits are extremely attractive. They must be willing to scrap the project if the costs grow, as is usually the case, and the benefits shrink, as is also usually the case, to the point at which it is no longer financially attractive, as is fortunately the case only occasionally. Finally, they must ensure that old systems are left intact until the new system has proven that it really works. I visited a company some years ago that demonstrated the validity of this last point. They had destroyed the old system; the new one had not worked for two months. The people in the plant were playing cards.

CONCLUSIONS

Systems do not implement themselves; people implement them. To succeed, a systems implementation effort must be managed effectively, by applying standard management principles (Planning, Organizing, and Controlling) with the intent to minimize risk. This is accomplished by using a time-phased commitment approach that provides management three separate opportunities to review costs and benefits and schedules, and to discontinue the effort with only the minimum possible resources having been expended at each of those decision points.

It is imperative that we, as MIS professionals, cause systems to be implemented properly in our respective companies. Our companies cannot afford the disaster of systems implementation failure. We cannot afford the continued negative publicity, and the resultant scepticism concerning our professional competence. Or, to be more blunt, a manager is only as good as his ability to deliver on his promises; we have proved for 20 years that we still lack that ability. It is time for us to acquire it, or face the consequences.

BIBLIOGRAPHY

Bliss & Laughlin Industries. Corporate Data Processsing Standards Manual. Oakbrook, Illinois.

Edson, Norris W., "The Realities of Implementing MRP," 23rd Annual Conference Proceedings (1980), American Production and Inventory Control Society, Inc., Washington, D. C.

Jones, Gary D., "Pitfalls to Avoid in Implementing MRP," 21st Annual Conference Proceedings (1978), American Production and Inventory Control Society, Inc., Washington, D. C.

Lasden, Martin, "Turning Reluctant Users On To Change," Computer Decisions, January 1891, pages 92-100.

McFarlan, F. Warren, "Portfolio Approach to Information Systems," Harvard Business Review, September/October 1981, pages 142-150.

Olsen, Robert E., "MRP Implementation — Doing It The User Way," 21st Annual Conference Proceedings (1978), American Production and Inventory Control Society, Inc., Washington, D. C.

Orr, Kenneth T., "Systems Methodologies for the 80s," Infosystems, June 1981, pages 78-80.

Salmere, Mitchel B., "How to Improve a Management Information System," Infosystems, November 1981, page 90.

Wight, Oliver. The Executive's New Computer, Reston Publishing Company, Reston, VA, 1972

APPENDIX A1

The Functional Specifications can include any and all of the following topics:

- General Background
- Management Summary
- Problem Definition
- Present System Description
- Major System Objectives
- Proposed System Description
- Economic Justification
- Detailed Plan of Action
- Responsibilities
- Proposed Schedule

APPENDIX A2

Functional Specifications for a system can include the following topics:

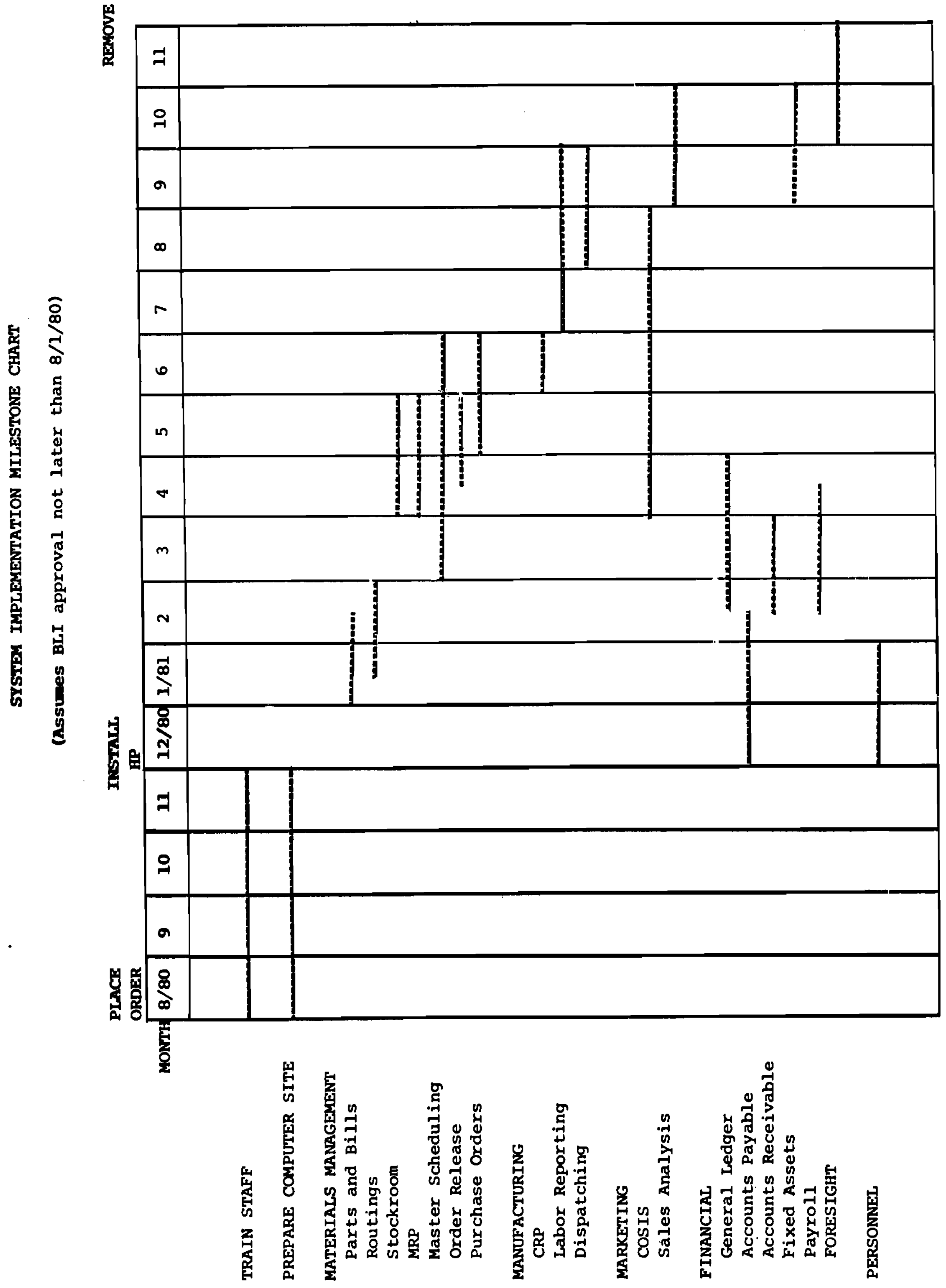
- Major Logic Chart(s)
- System Narrative
- Design Notes and Concepts
- Proposed Input and Output Layouts
- Proposed Controls
- Anticipated Throughput Volumes
- Future Capabilities
- Environmental Constraints on Expansion Capabilities
- Hardware and Software Considerations
- Proposed Training Plan
- Cost Considerations and Assumptions
- Interface Considerations
- Audit Considerations
- Contingency Plan

APPENDIX A3

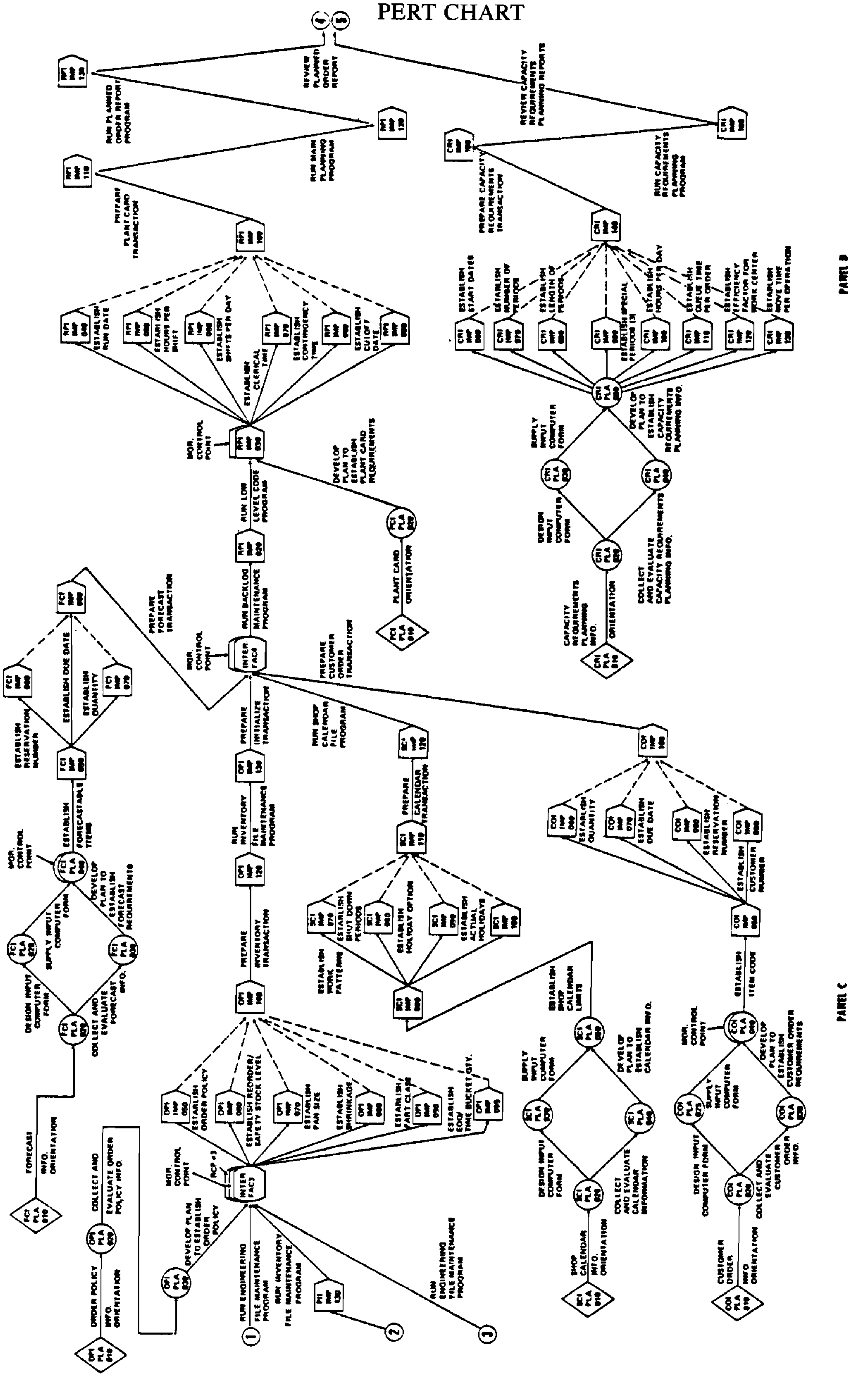
These items should be included in a Detail Design Specification; others may be added at your discretion:

- Detailed System Flowchart, defining input and output for each program
- Detailed Narrative for each section of the flowchart
- Program Run Sequences
- Audit Measures
- Quality Control Measures
- Internal Control Measures
- Security
- File and Data Conversion from the Present System
- Recovery Procedures
- Programming Conventions
- Test Specifications
- Test Standards
- Hardware/Software Environment
- Program Narratives
- Program Functions
- Program Specifications
- Program Logic
- Detailed Project Implementation Schedules

APPENDIX B
BAR CHART



APPENDIX B
PERT CHART



An Overview — Networking Cost Performance Issues

Russell A. Straayer
President, Data Communications Brokers, Inc.
Champaign, Illinois

The purpose of this paper is to give managers a conceptual overview of several key issues in datacommunications networking. We will focus on several practical, useful guidelines. At the end we will address what is practical and sensible today for most applications. As managers, you cannot wait for the promises of 10 years or even 5 years from now. A basic principle we will stress is response time for the terminal user.

Response time will be stressed because the major function of an HP3000 is to serve a human being, a person using the computer through a computer terminal. As human beings, we demand fast response times. We take our cars instead of waiting for the bus. We take the plane because the train or car is too slow for long distances. We eat at fast food restaurants. We read newspapers because we can scan vast amounts of information, pages and pages, in just seconds. We are equally demanding of fast response and convenience from our computers.

Getting to some practical information we can use today, we will first look at 10 specific topics, and cover them in rapid fire order. The 10 items could be full blown topics in their own right. What we as managers want to get out of the 10 points, however, are the practical guidelines. The technical staff folks and the vendors all have their technical pitches, and we must distill out sensible solutions that work today.

The 10 points, then are:

1. Telephone line cost trends
2. Data communications hardware cost trends
3. Technology trends of datacommunications hardware
4. Local networking alternatives
5. Packet protocol caveats
6. Satellite communications caveats
7. Importance of fractional second response time
8. Bits per second versus speed
9. Point to point versus multipoint cost/performance considerations
10. One large, efficient network

We will look at each one of these 10 points and highlight the important points using a few charts, graphs and illustrations. After we get through the 10 points, we will have a better basis for quickly getting to some firm network approaches. While there are many choices, many of them correct, most fall within a narrow range of practical solutions. In the end, it is then the responsibility of the manager to choose a solution that works. There will be no right or wrong decisions, just some that fit better than others. The ones that fit the best will be those made by management that has a good idea of the direction in which it is going and the destination it wishes to reach, and then asks the right questions to get most directly to the destination.

POINT 1. *The trends in telephone company communications line costs.* The trend in costs is up. Costs for lines are increasing at about 16 percent annually. We have an example of Illinois local private line rates. In 1970, a local private line cost just $3.50 per month, went to $15.00 by 1975 and is now up to $35.00 per month. That is for a line that may just go across a street. Telephone company charges for toll calls have not gone up much at all in the same time frame. Take note, however, that the phone company is slowly but surely working to put even local calls on a usage charge basis. The local call usage charges are already in place in Chicago, New York, Washington and other cities, both in the United States and elsewhere in the world.

The upward price pressure is clearly on dedicated facilities. Today, many areas in the United States have not yet caught up to Illinois, but will. You can look upon the Illinois example as a benchmark for where rates will go throughout the United States. The telephone companies are going to the state public utility commissions and gradually raising these private line rates.

Guidelines we can draw from this are:

1. Economize on lines using statistical multiplexers
2. Economize on lines using split stream modems
3. Be aware that local dial up lines can become much more expensive
4. Satellite and private local facilities may not yet be less expensive than telephone company private lines.

POINT 2. *Datacommunications hardware trends.* Users have been spared the full impact of the rising phone line rates by the reduction in the prices of datacommunications hardware. You are all probably familiar with the constant reduction in the prices of CRT displays and printing terminals. The prices for modems and multilexers, two key datacomm ingredients for on line networks, have also seen prices come down. In 1970, a 9600 bits per second modem cost about $10,000, or "a buck a bit." Today, some 9600 bps modems can be purchased for as little as $3,000. The prices of 4800 bps modems have dropped from $5,000 in 1970 to half or less than half of that price. Statistical multiplexers, a product just about 3 years old now, has seen a 25% price reduction in its young product life.

Guidelines we can draw from the hardware trends are:

1. Expect these approximate hardware costs;
   - 0 to 300 bps dial up modems now cost $200 to $300 per unit,
   - 1200 bps full duplex dial up modems, $700 to $900 per unit,
   - 2400 bps synchronous 201 type modems, $700 to $1,200 per unit,
   - 4800 bps sync 208 type modems, $2,000 to $4,000 per unit,
   - 9600 bps sync 209 or V.29 modems, $3,000 to $6,000 per unit,
   - 4 channel statistical multiplexers, $1,500 per unit,
   - 8 channel statistical multiplexers, $2,400 per unit,
   - Short haul synchronous modems, $600 per unit,
   - Short haul asynchronous modems, $300 per unit,
2. Expect prices to come down, features to be added, or both.

POINT 3. *Technology trends*. The technology trends of datacommunications hardware have also been at work to spare the user the full impact of rising phone line rates. The statistical multiplexer just mentioned is a perfect example. The stat mux, as it is called, has improved the price performance of ASCII CRT's and printers by more than a factor of two. The stat mux makes more efficient use of the phone line, lets the async ASCII terminal run faster than it could before, and often at lower costs than some slower, less efficient methods.

Another technology improvement of just a few years ago that we already take for granted is the 1200 bps full duplex dial up modem, equivalent to the Bell model 212. The 212 has been around for just about a half dozen years. Look for 2400 bps full duplex dial up equipment, at an affordable price, in the very near future.

The technology in datacommunications is making more efficient use of existing facilities, facilities are getting faster and more reliable, we are getting more control over the equipment, and more features. Microprocessors are showing up in more datacommunications equipment every day. We can do more and more for less and less. Just look at how vendors are scrambling to stay profitable in the face of this trend by giving you more and more features for the same prices, or even lower prices. That is good news for all users.

It is useful to look at the changes in technology over the years 1970, 1975, today, and what we may see in 1985.

1970

- Frequency division multiplexers (FDM)
- Time division multiplexers
- 103.113 type 300 bps modems
- 202 type 1200 bps modems
- 201 type 2000 to 2400 bps modems
- 208 type 4800 bps modems

1975

- Same as 1970 plus
- 212 type 300 to 1200 bps modems
- 209 type 9600 bps modems
- Short haul modems
- Coaxial cable modems
- First diagnostic tools

1980

- Same as 1975 plus
- Statistical multiplexers
- Integrated technical control systems
- Satellite
- Value Added Networks (VAN)

1985

- More software in datacomm products
- More features
- More cost effective local network products
- More cost effective fiber optics
- 2400 bps full duplex dial up modems

Guidelines drawn from technology trends:

1. Expect hardware to be very reliable, 20,000 to 50,000 hours Mean Time Between Failures (MTBF)
2. Look for simple to use features. For example, test functions that are useful but not too detailed if you or your staff do not use the functions daily. What good is it to know your bit error rate is 1 in 10 to the 6th, if you are not sure if that is good or bad.
3. As time goes on, look for more features for you money.

POINT 4. *Local networking*. Before we get to some of the details about local networking, take note that many local network products are still more promise than reality. Most installations today cannot yet take advantage of local networking technology because of the costs, or lack of interface compatibility.

Local networking alternatives include protocol options such as ethernet collision/detection, token passing methods, time division or frequency division. Links available include coaxial cable, twisted pair, infrared, microwave, and fiber optics. Hardware includes short haul modems, coaxial cable modems, ethernet type of interfaces, PABX's with data channel capability, and more hardware appears with increasing regularity. A good local networking overview article can be found in the December, 1981 issue of Datacommunications Magazine.

Guidelines concerning local networking:

1. For the HP3000, short haul modems are the most practical local network product.

2. Fiber optics are usually too expensive.
3. Ethernet type solutions are usually too expensive yet today.
4. Coaxial modems are often more expensive that short haul modems.
5. The short haul modem solution is practical today only within several miles of the computer. Beyond that, long haul modems are usually required.

POINT 5. *Packet Protocol Caveats:* Packet protocols that you hear about today include X.25, HDLC, SDLC, and many of the protocols found in the Value Added Network services such as Telenet and Tymnet. These protocols have a place, but are not going to be a complete solution. These protocols are:

- Designed mainly for message transfer, packets, electronic mail
- Too slow for full duplex operation
- Not suitable for the HP ENQ/ACK terminal to CPU handshake

As a practical guideline:

1. The packet protocols may have a place for mainframe to mainframe communications.
2. The packet protocols are rarely practical for terminal to mainframe communications.

POINT 6. *Satellite Communications Caveats.* The basic fact to consider abvout satellites is that they sit in a stationary orbit about 25,000 miles above the earth. It is that physical fact that contributes to the plus and minus features of satellites. First of all, satellite costs are not mileage sensitive. It matters not whether you go across the street or across the country, the cost is the same. Keep in mind that because it is not mileage sensitive, satellite is cost effective only on links of 500 miles or more.

A major consideration for satellite is local distribution of the data once you get it off of the satellite. From one major point to another major point, satellites can be cost effective, but not from many diverse points to many other diverse points.

Satellites are not very good for highly interactive data. Any transmission using satellite is bound by the speed of light, 186,000 miles per second. A round trip for data, via satellite, is 100,000 miles. Figure the round trip time to be almost 3/4 second.

Guidelines for satellite transmission:

1. Because of the round trip delay time, satellite is not suitable for the HP ENQ/ACK handshake or polled terminals (MTS).
2. You may wish to use satellite to connect mainframes, but not to connect terminals on line.

POINT 7. *Importance of fractional second response time.* At first glance, it may seem improbable that people need a 1/4 second response time or less to be efficient in interactive applications. It is in fact rather difficult to really grasp just how long 1/4 is.

To illustrate the importance of such small portions of time, consider the example of a radio call in show. We have all heard the person who calls in, and while listening to himself on a background radio, gets confused. The announcer says, "Please turn your radio down." Callers hear their own voices, fed back over the radio, but the delay disorients and confuses the caller.

To site another example, a user typing at a terminal will become very inefficient if the typed characters echoed back are delayed by just 1/4 second. The terminal user types ahead of the display rate of the characters and experiences what feels like a spongy keyboard. When a mistake is made at the keyboard, extra characters must be erased and retyped just to get back to the incorrect character.

When users are on line, say in a telephone order situation, it is important, if the person is to work at maximum efficiency, to get feedback for each keypress in 1/4 second or less. Typing at just 45 to 50 words per minute requires a key press every 1/4 second. When the keys do not get echoed back by the computer within that 1/4 second time window, the user slows down to match the echo time.

Keep in mind at this point that a high bits-per-second rate does not automatically mean fast response time. This fact can be easily illustrated by considering a 300 megabyte disk that we send by mail. If the post office delivers the disk in just 3 days, the transfer rate equals 9600 bits per second.

Guidelines for considering response time:

1. Response time over the communications link should be measured in milliseconds for interactive use. If the user operating a local terminal sees no delay, then the remote terminal user should see no delay, either.
2. For batch work, for file transfers and electronic mail, response time is less important than the bits-per-second transfer rate.

POINT 8. *Bits per second versus speed.* It may seem contradictory at first, but 9600 bits per second may not be as fast as 2400 bits per second. The difference is response time difference.

A Bell 2400 bps modem, model 201 allows for much faster polling in an MTS environment than does a 9600 bps modem, model 209. The difference in response time is accounted for in the Request To Send/Clear To Send delay functions of these modems. In a multipoint polled (MTS) environment, the RTS/CTS delay allows the modem at the central computer site to "tune" itself to the incoming signals from modems at any one of several remote sites.

The 2400 bps modem has an RTS/CTS delay of only 7 milliseconds, while the 9600 bps modem has a delay of 147 milliseconds. Given a typical poll of 12 characters and a 3 character response (a total of 15 characters), the

2400 bps modems allow for 3 times as many polls per second. For short message traffic, the lower speed modem may be a good bit faster.

Guidelines regards bits-per-second versus speed:

1. In a polled (MTS) environment, 2400 or 4800 bps is often the best you can do for your money.
2. File transfers to an RJE station or mainframe to mainframe can benefit from the 9600 bps modems, since there is usually no concern about the RTS/CTS delay.
3. Using statistical multiplexers and asynchronous terminals, the typical best speed is 2400 bps for the terminals, and 4800 bps for the composite modem link, for up to 8 terminals. 9600 bps may be called for if printers are heavily used.

POINT 9. *Point to point versus multipoint (MTS).* In an HP3000 environment, point to point usually means using asynchronous CRT's and printers. Multipoint is the MTS environment. What we want to do here is look at the differences from the datacomm point of view.

We have a case study to look at comparing MTS with point to point using statistical multiplexers. The user is in Dundee, Michigan. The test involved 2 CRT's and 1 printer, running on 4800 bps Bell 208 modems. The specific test was to evaluate how the user saw response time, and to measure actual output volumes.

The results were clearly in favor of the statistical multiplexer method of operation, even when terminals were slowed from 4800 under MTS to 2400 bps async. Measured output was more than double for the async mode of operation. The users at terminals saw noticeable reduction in response time when the printer was running, but not when using the asynchronous mode and statistical multiplexers.

It should be noted, too, that the async terminal operation is usually easier to set up, easier to diagnose, easier to maintain.

Guidelines on point to point versus multipoint:

1. In most cases on an HP3000, point to point asynchronous operation proves to be the most cost and performance effective.
2. You may wish to consider MTS (polled terminals) if you do very little printing along with CRT displays at remote sites and your response time can reduced by a few seconds. The printer is the major consideration.

POINT 10. *One large efficient network.* The network we will examine here is the United States switched phone network, the one we use when dialing local or long distance calls. The US phone network, managed primarily by AT&T, is well developed, efficient, and employs the princples of good networking.

Examining the routing of a phone call from Champaign, Illinois to San Antonio provides a good look at the structure of the network. A typical call is routed:

1. From the local telephone station, over a station loop to the central or end office.
2. The end office connects to a toll office via a toll connecting trunk.
3. The toll office, say in Champaign, connects to another toll office via an intertoll trunk. There are several classes of toll offices in the network hierarchy.
4. To avoid going through the entire chain of toll office command, the call may be routed from one lower level office to another, close to the destination. The lower level offices are connected via high useage trunks (HUT's).
5. The toll office close to or actually in San Antonio connects the call to the end office in the city, which rings the local phone.
6. The call is completed when you pick up the phone.

This entire process is referred to as circuit switching, since the call connection uses actual, physical circuits. Packet switching, on the other hand, does not make a connection via actual circuits, but packages up the data and routes the databased on destination addresses included in the packages.

Suggested guidelines based on the phone network:

1. Think of your HP3000 as though it is a PABX on location in a business. Its purpose is to connect terminals to files and terminals to terminals.
2. Terminals will almost always be connected to the HP3000 the way phones are connected to a PABX or central office, with one port per terminal, just as there is one physical line per phone number or extension.
3. Access through the HP3000 should be as standard and simple as possible.

At this point, we will look at a specific network on an HP3000. The user here is Johnson and Staley of Nashville, Tenessee. This network takes all this information and illustrates the kind of datacommunications most practical for 90% of all HP3000's.

The Johnson and Staley application is on line order entry and inventory maintenence for a distributer of school supplies.

The Johnson and Staley application embodies our 10 points in the following ways:

1. The line configuration is designed to keep the phone line costs to a minimum.
2. The links between the multiplexers are 4800 bps, about the best in price for the bits-per-second rate needed. Five years ago Johnson and Staley would probably have decided that the modem and multiplexer costs were to high to go on line.
3. The DDS, bandsplitting of DDS, and the stat mux's are all late 1970's technology.
4. Local networking is not applicable here.

5. Satellite links are not cost or performance effective here.
6. Packet networks are not cost or performance effective here.
7. The on line order entry activity required very fast response times.
8. A link of 9600 bps per terminal grouping would not improve performance in this application. The volume of data is small for this application. The need is for instant access to the inventory and order records.
9. Line cost savings that might otherwise be available only thru multidrop networking are acheived by bandsplitting.
10. Terminals are connected on a per port basis to the HP3000, much like extensions to a switchboard.

SUMMARY OF HOW 90% OF ALL HP3000'S COMMUNICATE

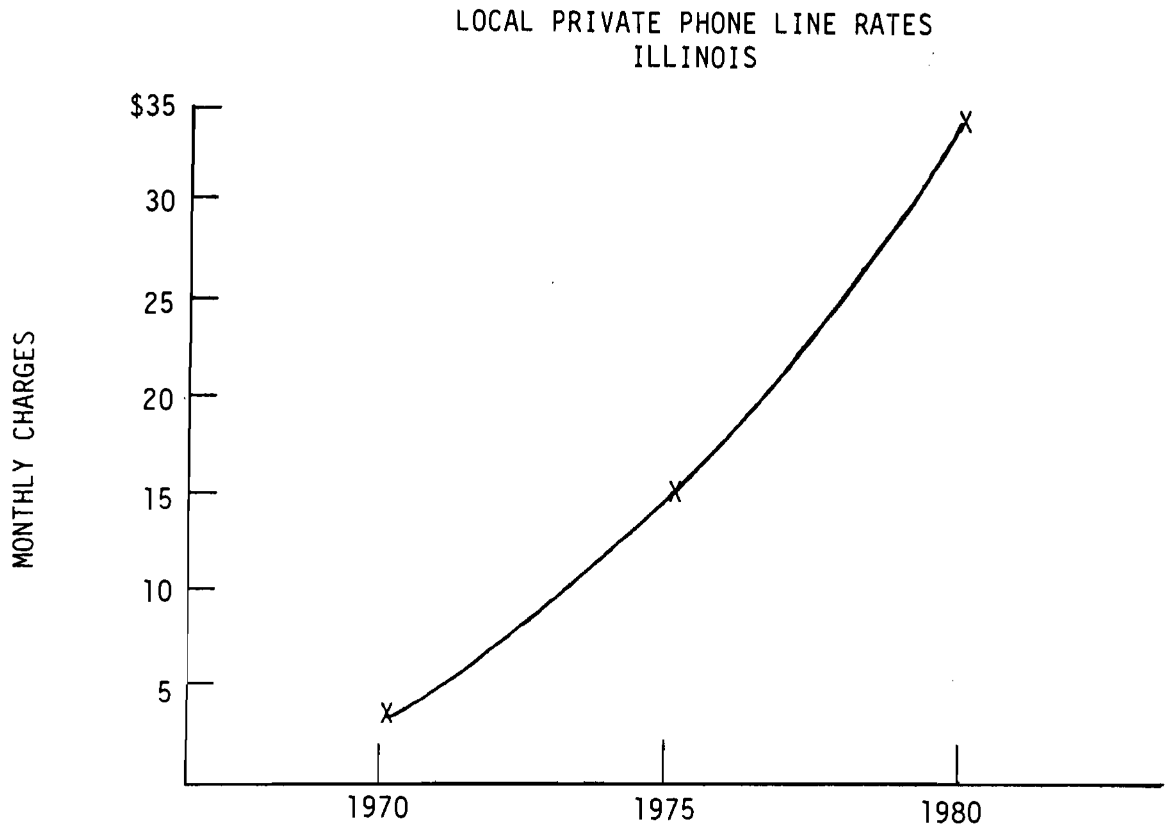
HP3000 to HP3000 or larger mainframe:

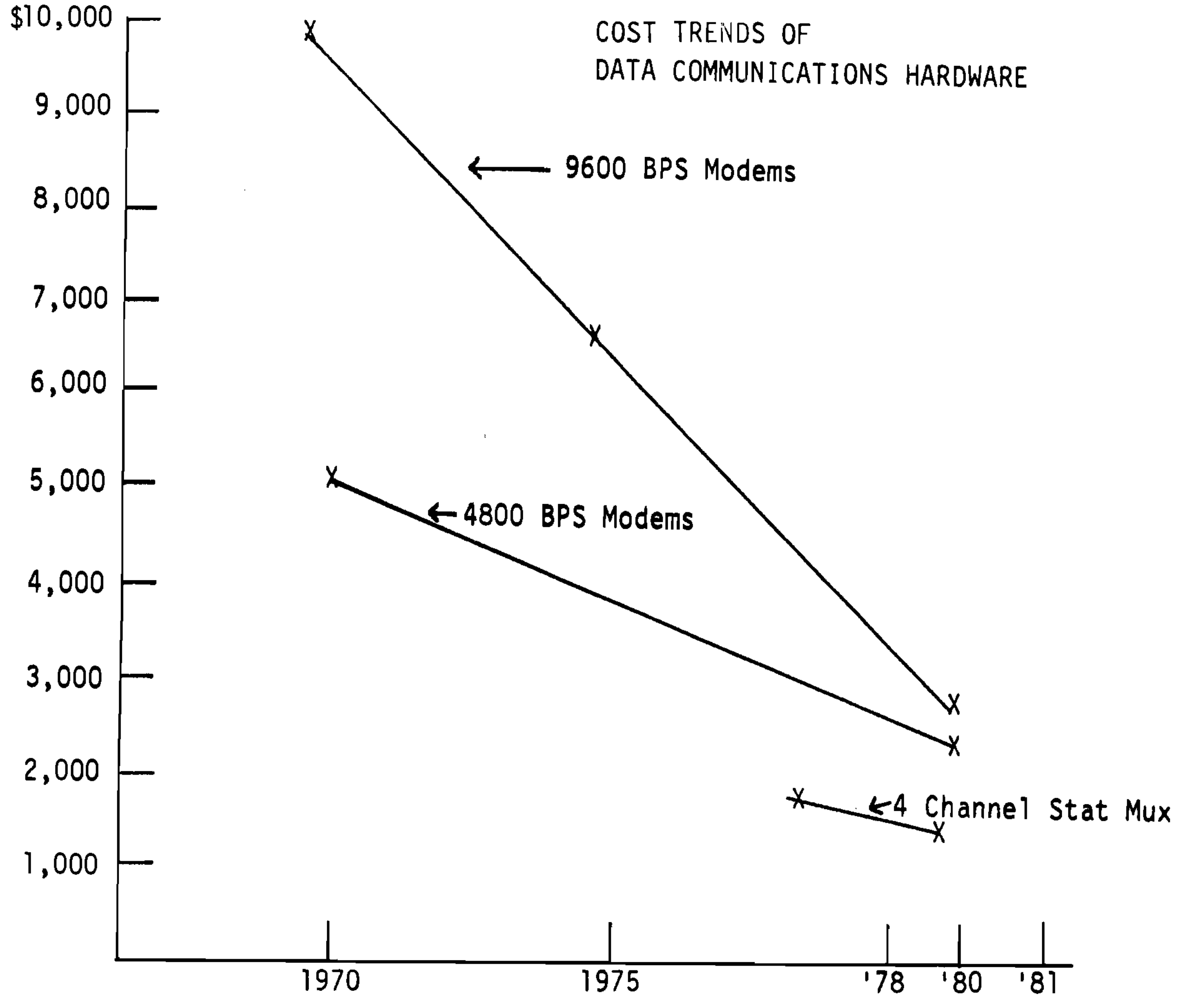
- Synchronous facilities
- Private line
- 2400, 4800 or 9600 BPS
- Digital Data Service (DDS)
- Very few satellite links

HP3000 to terminals:

- Hardwired or within 100 miles of the mainframe
- Asynchronous, with dial up, single modems or stat mux's
- Speeds of 1200 or 2400 BPS

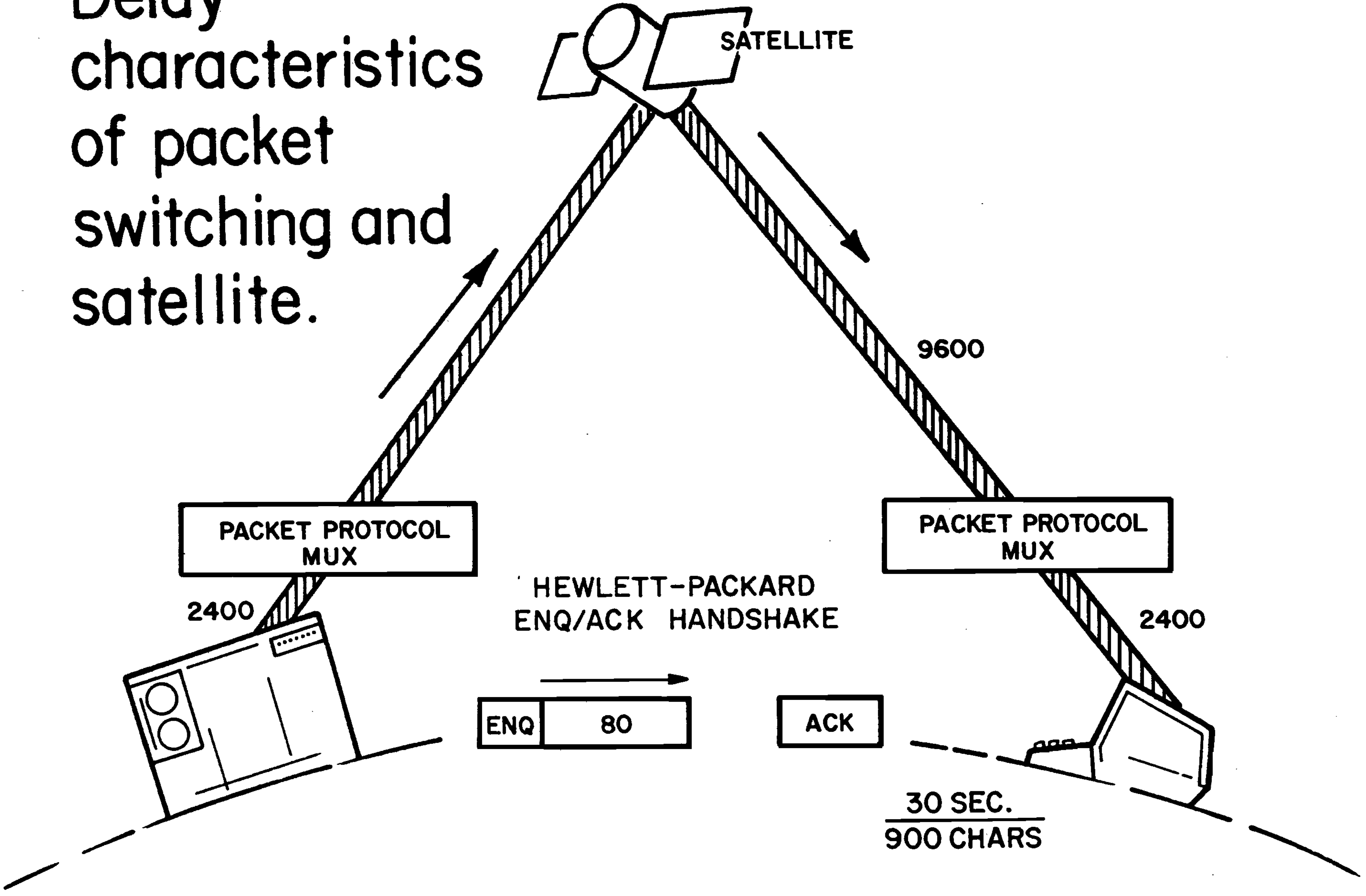
In conclusion, for all the choices and possible confusion surrounding HP3000's in a data communications environment, 90% of all systems have the same configurations, with minor variations.


LOCAL PRIVATE PHONE LINE RATES
ILLINOIS
MONTHLY CHARGES
$35
30
25
20
15
10
5
1970
1975
1980

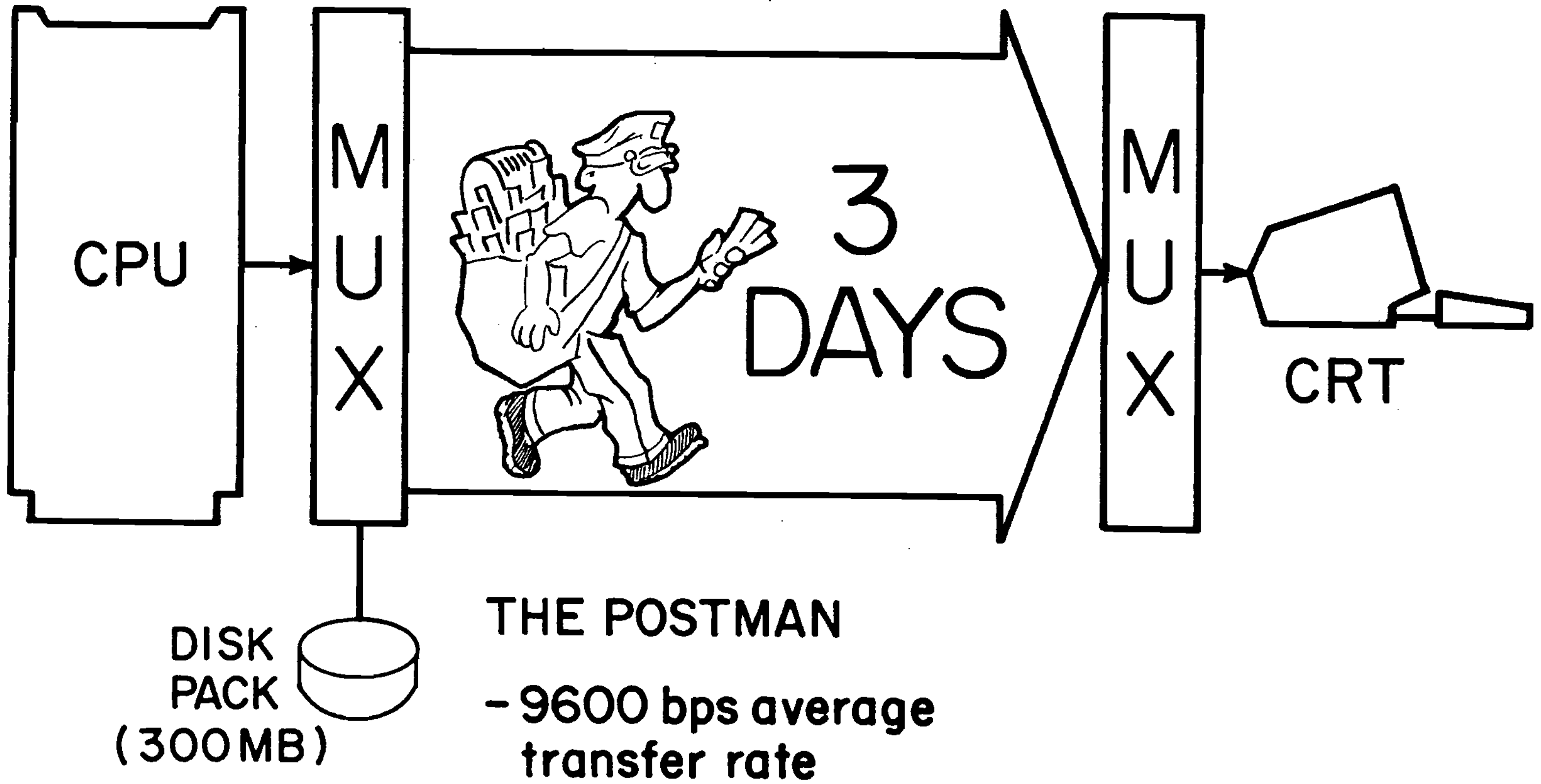

COST TRENDS OF
DATA COMMUNICATIONS HARDWARE
$10,000
9,000
8,000
7,000
6,000
5,000
4,000
3,000
2,000
1,000
9600 BPS Modems
4800 BPS Modems
4 Channel Stat Mux
1970
1975
'78
'80
'81

C.C.I.81

Delay characteristics of packet switching and satellite.



The Postman delivers 9600 bps.



THE POSTMAN

- 9600 bps average transfer rate
- Good for large volume several day delivery

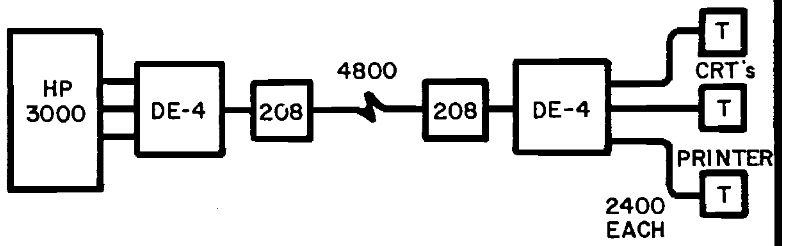
C.C.I. 81

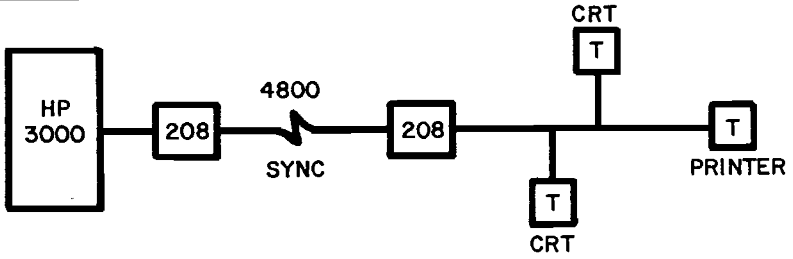
C.C.I. 81

BENCHMARKED 12-79

2 network alternatives

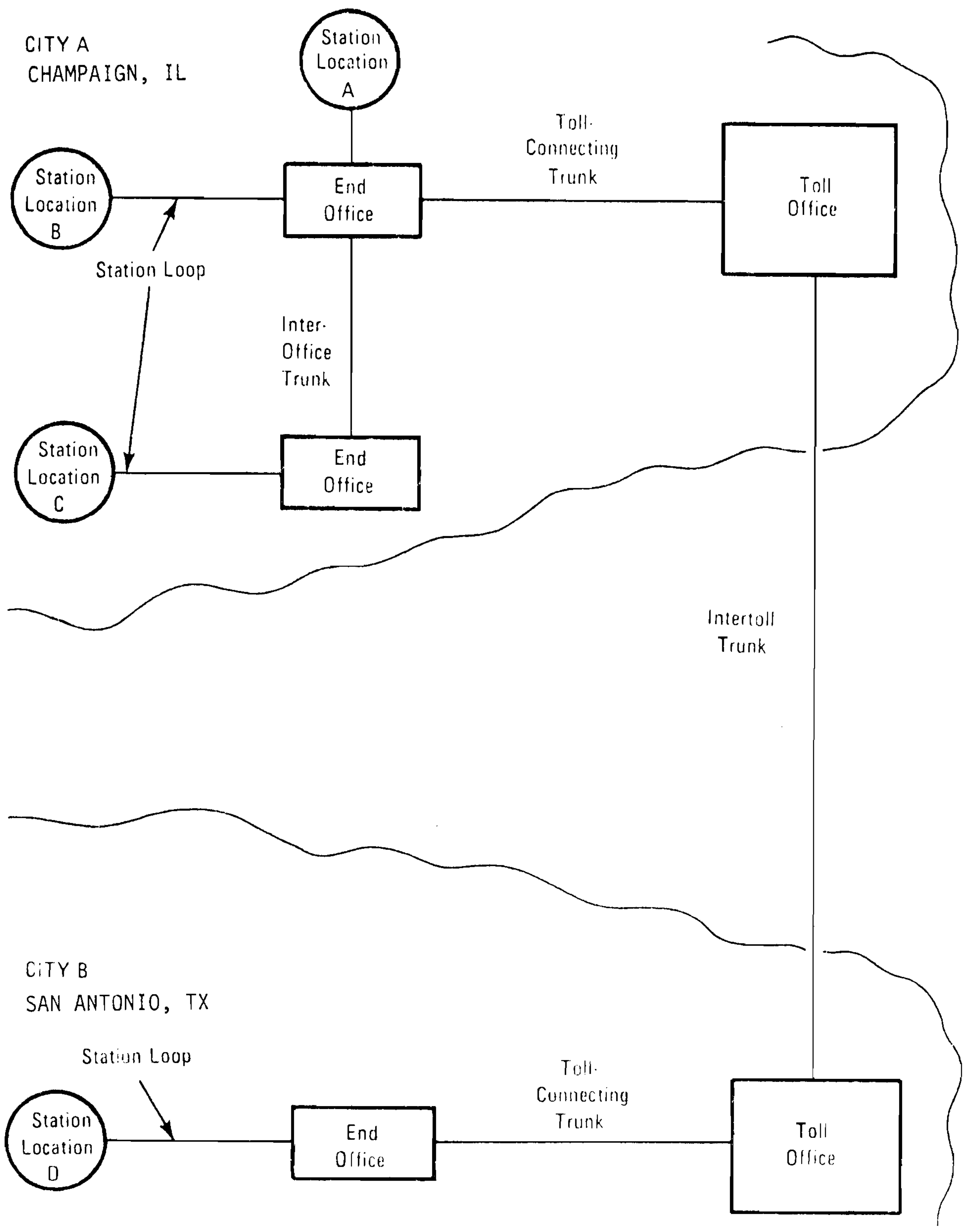
Asynchronous stat muxed terminals	Polled
POINT TO POINT	**MULTIDROP/ MULTIPOINT**



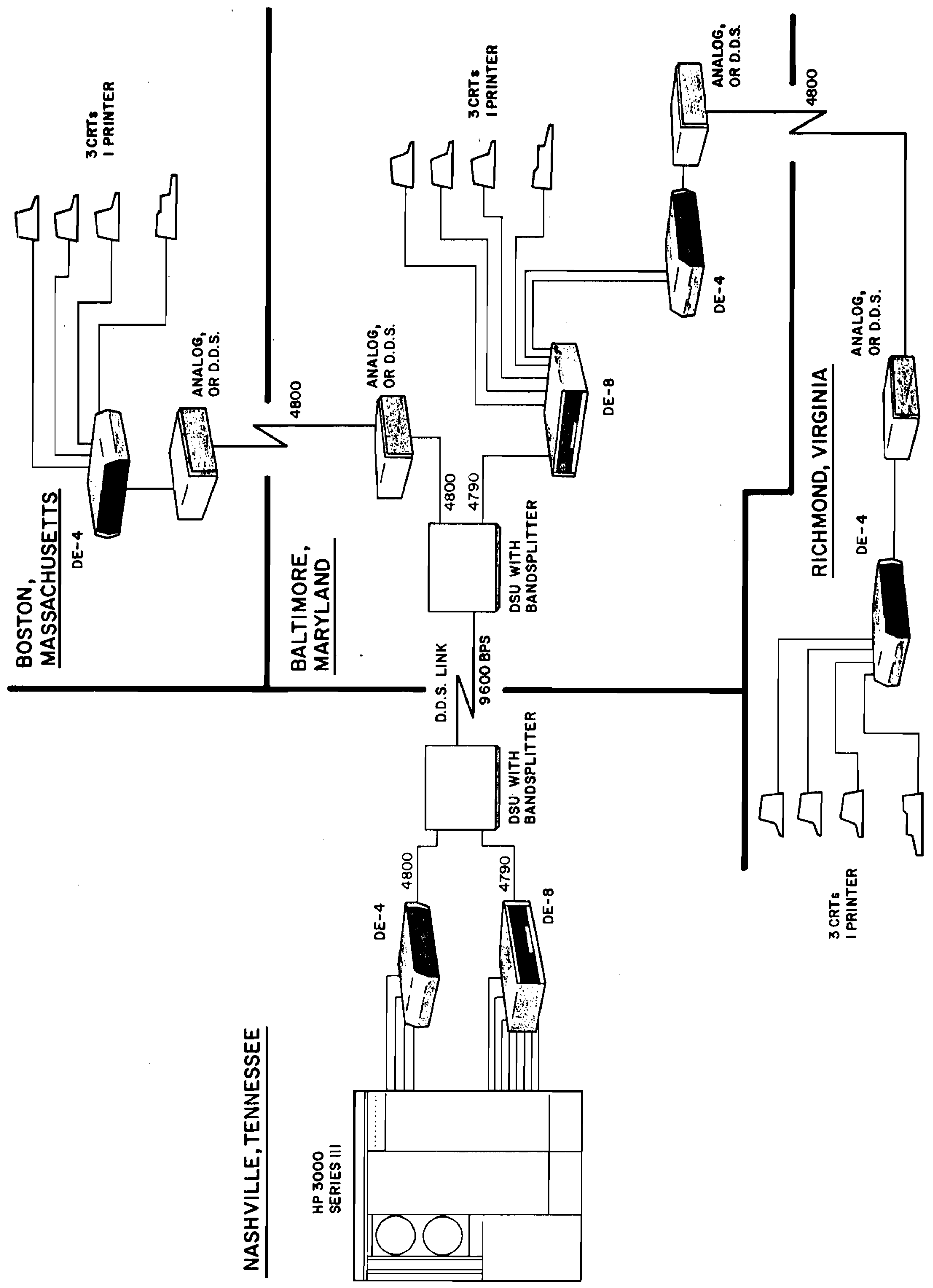


TESTED AT DUNDEE CEMENT, DUNDEE, MICHIGAN

Point to point	Multidrop/Multipoint
Total one way output...	Total one way output...
listing only —	listing only —
280 CPS Average	115 CPS Average
Same modems, terminals, data output – ASYNC Interactive	Same modems, terminals, data output – SYNCHRONOUS Polled



Typical Routing for Connections


NASHVILLE, TENNESSEE
HP 3000 SERIES III
DE-4
DE-8
4800
4790
DSU WITH BANDSPLITTER
D.S.S. LINK
9600 BPS
BOSTON, MASSACHUSETTS
DE-4
ANALOG, OR D.D.S.
4800
3 CRTs 1 PRINTER
BALTIMORE, MARYLAND
ANALOG, OR D.D.S.
4800
4790
DSU WITH BANDSPLITTER
DE-8
DE-4
3 CRTs 1 PRINTER
ANALOG, OR D.D.S.
4800
RICHMOND, VIRGINIA
ANALOG, OR D.D.S.
DE-4
3 CRTs 1 PRINTER

Software Management Techniques

Janet Lind

There is currently much information available to document the fact that the cost of hardware is decreasing dramatically, but the cost of software continues to climb. When questioning the source of this problem, it is necessary to consider that many hardware functions are now being implemented in software or firmware. It is also true that computers are constantly being used in new applications, and computer users have increasingly sophisticated needs.

Today's software systems suffer from a variety of problems. Often they are delivered later than originally scheduled. The systems may cost more than the original projections. The software may not meet the user's requirements, or may be unreliable. When the need arises to correct or upgrade the system, the cost involved may be in excess of the cost of the original system.[18]

One of the most pressing problems in software project management is the lack of a well-developed structure for guiding the individual programmer. Instead of directing the programmer's activities, the manager can often only manage an idea until all parts of the project are completed. This problem arises from the fact that the only clearly defined point in the programmer's work is completion. More definition of the process is needed.[2]

There is no reason why software development should be exempt from the formats found in other engineering fields. Lab notebooks, design reviews, and failure and reliability analysis have proved their value.

The lack of a disciplined approach to software development may produce programs which are difficult to understand or maintain, affecting overall cost. Therefore it is important to develop a more rigorous framework to delineate the several steps in the programming process. Knowing the proper steps to follow will allow a programming team to develop more common objectives about the problem solution. This will improve the product and the group motivation by allowing the programmers to focus on more immediate goals.

Even though the approach being taken is to define a series of programming steps, it is always important to allow feedback to improve the product. A sequential description of program development steps will be defined here, but a problem found may cause a redefinition in preceeding steps to provide a more correct solution.[8]

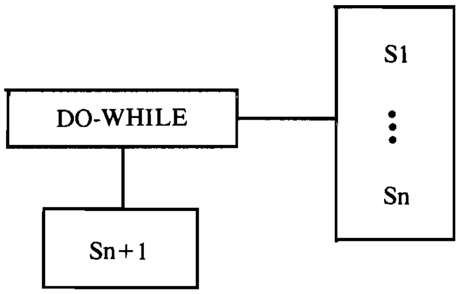
When first approaching a software project, it is necessary to perform a problem analysis. Here the inputs and outputs must be specified and the relationships between them must be described. A programming notebook should be kept to indicate how decisions were reached.[4]

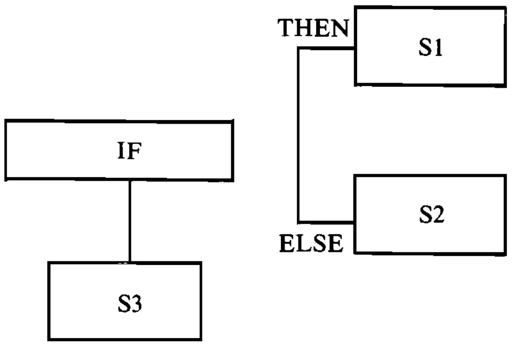
Part of the problem analysis includes decisions about the resources available. This includes both people and computer power. When considering the hardware used, it is no longer strictly correct to consider implementing everything possible in software. To increase productivity and to simplify code requirements, it may be worthwhile to purchase or develop hardware to meet the problem.

Another choice would be the use of multiple processors, which gives greater flexibility than implementing a function in hardware. This could also decrease the complexity of a given program, for it will no longer be responsible for as many portions of the function. Programs could also run concurrently, reducing timing constraints on a single system. This would make programming in higher level languages more attractive because the added processor capabilities offset the less efficient code produced. After completion of problem analysis, a walkthrough should be performed.

The solution design is driven by the I/O and their relationships defined in the problem analysis. Several different documentation techniques and evaluation criteria can be used in the structured design. Data flow diagrams can be used at the high level abstraction to model the flow of data through the system.[5]

Higher order software notation, or HOS, which was developed as part of the Apollo Program at Draper labs, defines a very useful flowcharting technique. Each control structure has a horizontal block showing the program flow in that structure. This type of flowchart does not show extra arrows, and allows easy identification of each possible branch. This notation also uses the same identation as should appear in the actual code.[3]





HOS Example

Some of the evaluation criteria used in structured design include decisions about the possible program development tools available. Certain programming languages may provide better support for the data structures to be used. They may also affect the amount of coupling required between modules. It is important to consider the capabilities of the computer system on which the program will be run, including memory management techniques and I/O capabilities.

When doing structured design, the design team is often tempted to perform just the top level abstraction as a team, designing the lower levels individually. There are some important reasons for doing a single integrated design of the entire application. First, subdivision of the design may result in excessive coupling of the major systems. The resulting packaging into programs from a subdivided design may be suboptimal. A complete overall structural design could produce more efficient and convenient packaging. Subdividing the design work will very often result in duplicate programming. It is particularly unfortunate when minor changes occur in a few structures, yielding a new system which could have shared entire subsystems and many levels of modules.[9]

Even though there are reasons for completing the entire structural design as a single unit, this is not always possible. In that case it would be best to produce a high level abstraction of the program flow and identify the more independent subsections. Those with few, uncomplicated interconnections could be treated independently. To avoid duplication of code, frequent mutual design walkthroughs and cross-checks should be performed.

Either while the structured design is being developed, or after its completion, the testing must be planned. It is necessary to design the test cases before the coding is begun. This allows peer review to verify that the designed code can be tested.

If the HOS flowchart notation is used, each program branch can be easily identified, and therefore tests can be designed to exercise each branch. If each program branch is numbered, a test matrix can be developed to indicate which tests execute which branches. The input and output to each test must also be specified.[4]

Both the structured code design and the test design should be carefully reviewed via structured walkthrough techniques. When considering walkthroughs, it is necessary to determine if it is more economical for an error to be found by the programmer, or by a group of 3 to 5 people. Part of the cost-benefit calculation is the turnaround time for repairing errors. Recent studies indicate that it is roughly ten times more expensive to fix a design error after it has been coded than to repair an error detected in design phase. It is also quite possible that when looking for errors, the programmer can repeat a logic error and never find the bug. Walkthroughs can help avoid this.[10]

Test Case	Input		Branch					Output		
	V1	V2	1	2	3	4	5	V1	V3	V4
1	0	0	X					0	0	0
2	0	1		X	X			0	0	2
	1	0		X		X		0	2	1
3	1	1	X		X		X	1	1	1

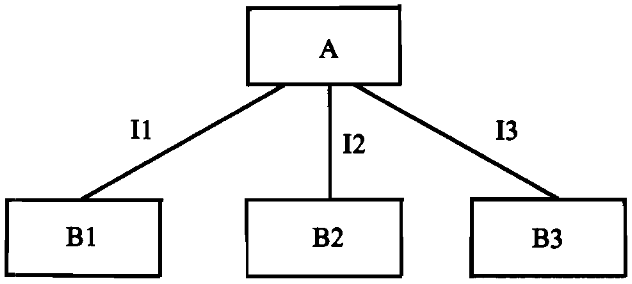
Test Matrix Example

There are other walkthrough benefits which must be weighed against the cost. The product quality is improved. The walkthrough participants are better trained in the product and are able to exchange important information. This exchanged information increases the probability that the product can be salvaged if a programmer leaves before completion. A walkthrough is also a good environment for feedback into other areas.

After the designs have been accepted, coding and debugging can begin. Here structured programming techniques should be both understood and applied. Using the HOS flowchart technique makes program flow and structuring obvious at coding time.

It is too simple to believe that code without "GO-TO" commands is always good. The language being used should be well comprehended by the programmers to ensure that the proper constructs are used. The code within each module must be structured. Concurrent documentation should also be kept.

With developing and testing code, it is also necessary to choose between a top-down or bottom-up approach to the overall structure. If hardware is being developed concurrent with software development, the lower level modules may be needed first to verify the hardware. In most other cases, a top-down approach can provide a more obvious visual presentation. This technique also allows modules to be tested together sooner. The interface between a node and its predecessor can be tested as soon as the lower level node is developed, allowing design or implementation errors to be detected and corrected earlier.[9]



Example

TOP-DOWN	BOTTOM-UP
1. Code and debug A	Code and debug B1
2. Code and debug B1	Code and debug B2
3. Test I1	Code and debug B3
4. Code and debug B2	Code and debug A
5. Test I2	Test I1
6. Code and debug B3	Test I2
7. Test I3	Test I3

A librarian function is helpful during coding and testing. The librarian can be an appropriately trained person, or an automated system. The librarian should maintain source programs and listings, as well as organizing all other technical information.

An automated system would avoid mixing media, which could be helpful in keeping a very accurate record of what changes are made. A record kept during edit phase could record what lines were modified and which variables were affected. A time stamp on this information could help other programmers know which version of code they were using. The knowledge that this system is being used will encourage a programmer to carefully analyze each change.

When the code can be tested, the test case matrix should be used to direct the tests applied. It may be useful to have the test run by the librarian. The test results should match those predicted, and a run log should be kept to document the test results. The purpose of the run should be stated, followed by an analysis of the run in terms of that purpose. This allows feedback for code correction and avoids haphazard modification. Any corrective actions which must be taken by the programmer should also be recorded.[5]

It may also be valuable to keep a time log to summarize the time needed for each step. This forces the programmer to review the actual effort expended in a task, and helps for making more realistic future estimates.

Throughout all activities, an independent auditing function can be performed. This will help detect errors unnoticed by the development team, and provides feedback.

The system described here is relatively involved and may be difficult to implement all at once. A pilot project could be chosen to use structured coding, structured design, and informal walkthroughs. As the process is implemented, it may be valuable to measure certain aspects such as the number of debugged lines of code produced per day and the number of bugs found after release. This can aid in future estimates. The amount of time spent in each walkthrough and the number of bugs found there should also be measured to help improve the techniques used.[6]

BIBLIOGRAPHY

[1]F. T. Baker, "Chief Programmer Team Management of Productin Programming," *IBM SYST. J.*, vol. 11, No. 1, 1972.

[2]F. T. Baker, "Structured Programming in a Production Environment," *IEEE Trans. Software Eng.*, pp. 241-252, June 1975.

[3]M. Hamilton and S. Zeldin, "Higher Order Software — A Methodology for Defining Software," *IEEE Trans. Software Eng.*, vol. se-2, pp. 9-32, Mar. 1976.

[4]P. Hsia and F. Petry, "A Framework for Discipline in Programming," *IEEE Trans. Software Eng.*, vol. se-6, no. 2, pp. 226-232, Mar. 1980.

[5]P. Hsia and F. Petry, "A Systematic Approach to Interactive Programming," *Computer*, pp. 27-34, June 1980.

[6]M. Page-Jones, *The Practical Guide to Structured Systems Design*, Yourdon Press, New York, N.Y., pp. 267-284, 1980.

[7]C. H. Reynolds, "What's Wrong with Computer Programming Management?," *On the Management of Computer Programming*, G. F. Weinwurm, Ed., Auerbach, Philadelphia, Pa., pp. 35-36, 1971.

[8]M. Walker, *Managing Software Reliability – the Paradigmatic Approach*, A. Salisbury, Ed., North Holland, New York, N.Y., pp. 32-41, 1981.

[9]E. Yourdon, *Managing the Structured Techniques*, Prentice-Hall, Inc., Englewood Cliffs, N.J., pp. 10-88, 1979.

[10]E. Yourdon, *Structured Walkthroughs*, Prentice-Hall, Inc., Englewood Cliffs, N.J., pp. 87-100, 1979.

Structured Analysis

Gloria Weld
Hewlett Packard Corporation

In any programming project, there are three areas of partition: Analysis, Design, and Implementation. All three of these areas can benefit from a systematic, structured approach.

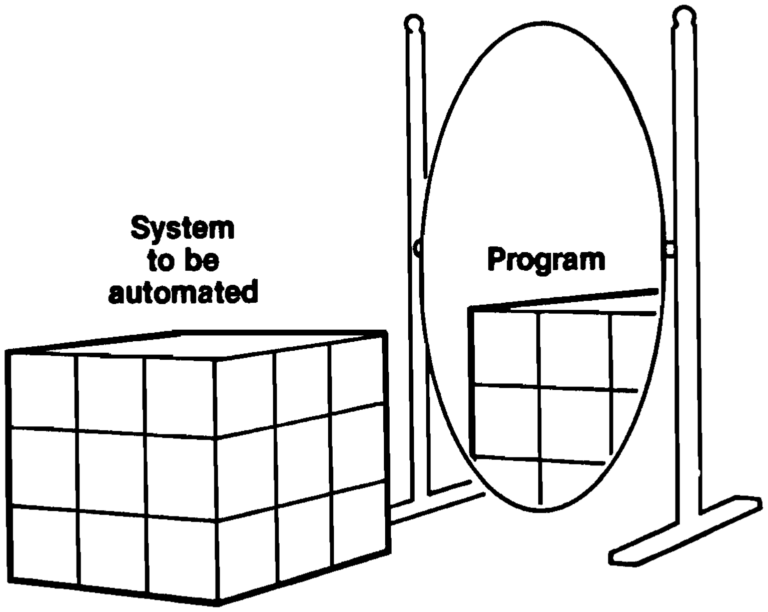
Today we will discuss Structured Analysis. In our discussion, our underlying assumption will be that we are called upon to design (automate) a new system in order to replace an existing system.

ALL OF US WHO ARE INVOLVED WITH PROGRAMMING AND PROGRAMMERS ARE CONCERNED WITH MAKING SURE THAT THE CODE WHICH IS WRITTEN ADEQUATELY AND APPROPRIATELY REPRESENTS THE SYSTEM WHICH IS TO BE AUTOMATED.

STRUCTURED ANALYSIS IS A METHOD TO ACHIEVE THAT GOAL.

Our Goal:

The program written must truly represent the system to be automated.



SOME TOOLS OF STRUCTURED ANALYSIS

O DATA FLOW DIAGRAMS (DFD'S)

O DATA DICTIONARY

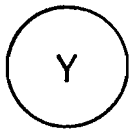
O STRUCTURED ENGLISH

DFD NOTATION

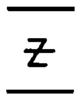
1. DATA FLOWS, REPRESENTED BY NAMED ARROWS

$$\xrightarrow{X}$$

2. PROCESSES, REPESENTED BY NAMED CIRCLES I.E. ("BUBBLES")



3. FILES, REPRESENTED BY NAMED STRAIGHT LINES



4. DATA SOURCES AND SINKS, REPRESENTED BY NAMED BOXES

$$\boxed{A}$$

DFD'S

I. A LANGUAGE

II. AN EXCELLENT TECHNIQUE FOR UNCOVERING MISUNDERSTANDINGS DURING THE ANALYSIS PHASE OF A PROJECT.

COMMENTARY

HOW DO YOU ANALYZE A SYSTEM?

YOU TALK. YOU TALK TO THE PEOPLE WHO ARE PART OF THE SYSTEM. YOU ASK THEM WHAT IT IS THAT THEY DO.

Discussing "how things work" with a participant in a system can often lead to confusion. Quite naturally, there are multiple views of the system. Each participant in the system views the situation from his own vantage point. Thus, analysis derived from discussion with one participant will often conflict with analysis derived from discussion with another participant.

For example:

DESCRIPTION OF A "HOSPITAL SYSTEM"

A PATIENT COMES INTO THE HOSPITAL AND CHECKS IN.
IF HE IS REALLY SICK, HE DOESN'T CHECK IN HIMSELF
BUT HE'S PUT INTO A WHEELCHAIR AND SENT RIGHT UP
TO A ROOM (UNLESS HE'S AN EMERGENCY-ROOM PATIENT).
THEN THE DOCTOR ORDERS ALL THE LAB TESTS HE NEEDS.

SOMETIMES MATERNITY PATIENTS GO TO THE LABOR-DELIVERY
PART OF THE HOSPITAL RIGHT AWAY.

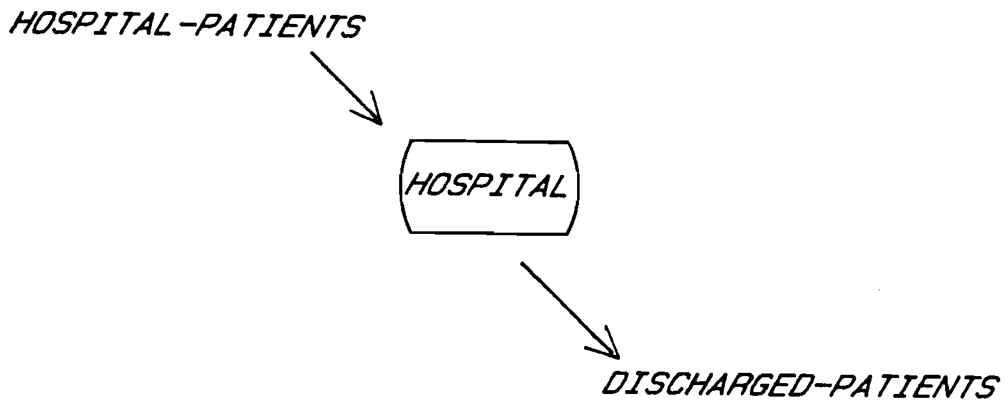
EMERGENCY ROOM PATIENTS HAVE TO WAIT IN THE EMERGENCY
ROOM UNLESS THEY NEED TRAUMA CARE RIGHT AWAY.

AFTER TESTS AND X-RAYS ARE TAKEN, COPIES GO INTO THE
CHART AND A COPY GOES TO MEDICAL RECORDS.

I AM CHIEF COOK IN THE HOSPITAL KITCHEN. ALL FOOD
GOES THROUGH ME. EVERY DAY WE COOK BREAKFAST, LUNCH
AND DINNER. WE COOK SPECIAL FOODS FOR PEOPLE WHO ARE
ON SPECIAL DIETS, TOO. THAT'S THE HARDEST PART!

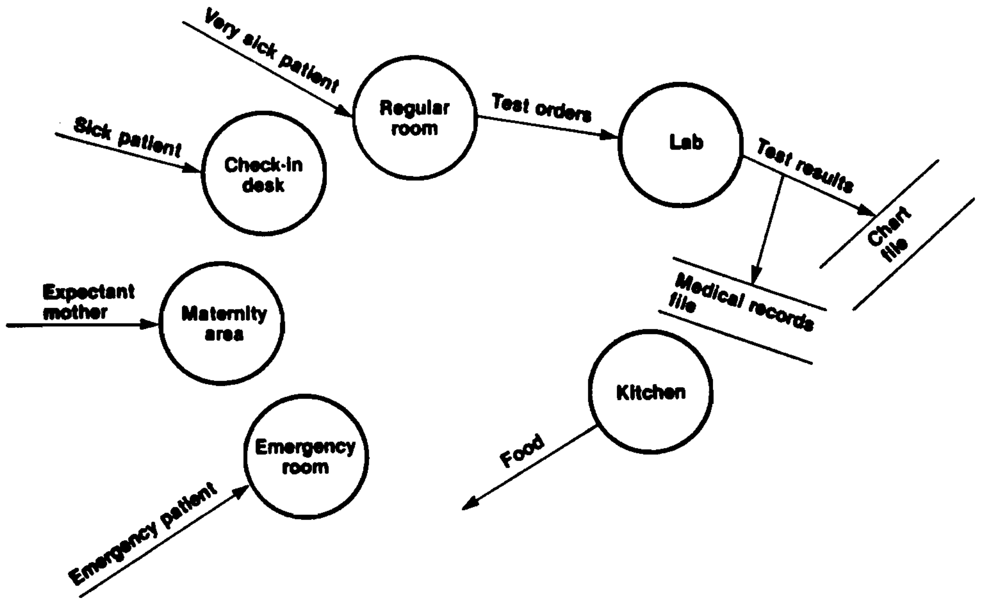
From this description, we can derive a "Top Level" of analysis:

HIGHEST LEVEL
OF
"HOSPITAL SYSTEM"



A "First Pass" DFD representing the HOSPITAL SYSTEM might look like this:

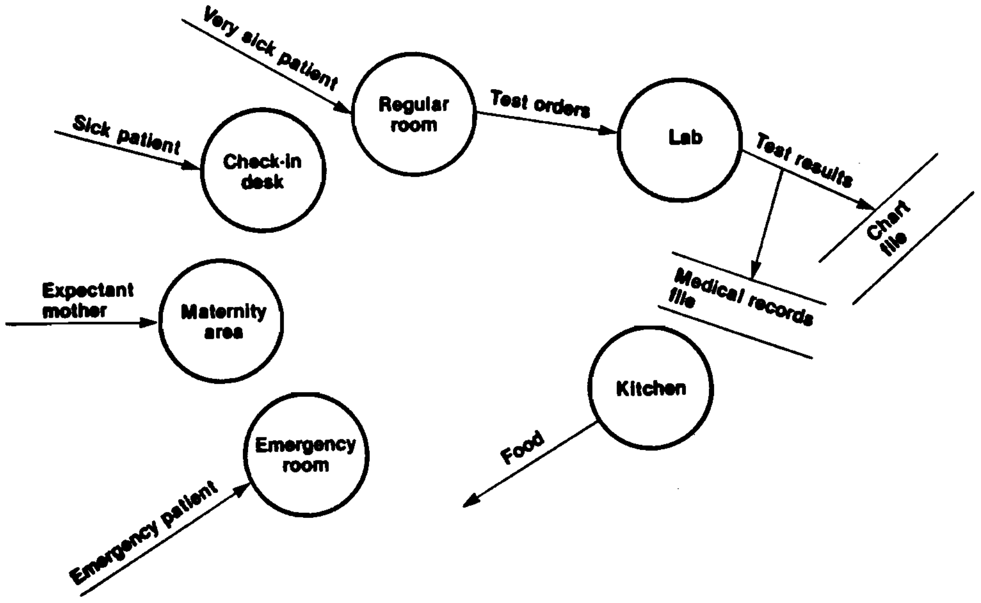
DFD of "Hospital System" Pass I (Taken from Verbal Report of a Participant)



As you can see, there are many empty spaces in our first pass DFD. From the description given us by our participant, we have created a DFD with data-flows entering process bubbles and no data exiting. We also have data flows coming out of process bubbles where no data ever entered.

Our "Tests for Correctness" which point out an incorrect DFD immediately point out to us that our understanding of this system is conceptually incorrect. And we (for the most part) know exactly what it is we don't understand.

DFD of "Hospital System" Pass I (Taken from Verbal Report of a Participant)

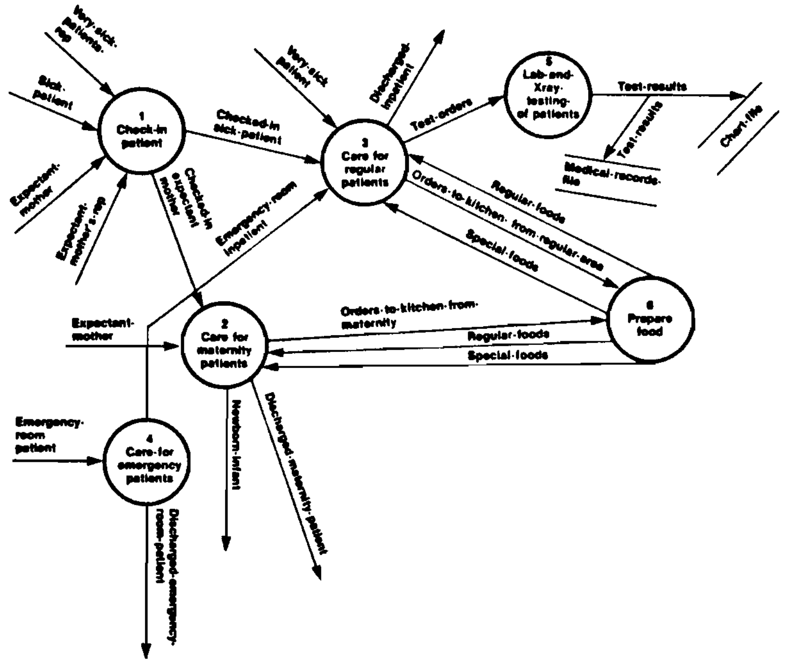


QUESTIONS WHICH COME UP WHEN TRYING TO ANALYZE THIS PASS 1 DFD

- WHAT HAPPENS TO A PATIENT WHO IS NOT VERY SICK? AFTER HE CHECKS IN, WHAT DOES HE DO?
- DOES A PATIENT WHO IS TOO SICK TO CHECK IN HIMSELF EVER GET CHECKED IN?
- DO EMERGENCY ROOM PATIENTS WHO DON'T NEED TRAUMA CARE EVER GET OUT OF THE EMERGENCY ROOM?
- HOW DOES A PATIENT (EITHER A REGULAR PATIENT, MATERNITY PATIENT, OR EMERGENCY ROOM PATIENT) EVER GET OUT OF THE HOSPITAL?
- HOW DOES THE KITCHEN KNOW WHAT SPECIAL FOODS ARE NEEDED? WHERE DOES THE FOOD GO ONCE IT LEAVES THE KITCHEN?

After asking those questions, we come to a DFD like this. True, it appears confusing. However, it is a pictorial representation of our system, a tool for discussion between the analyst and the participant.

DFD of "Hospital System" Pass II



We might wish to expand one of our process bubbles, in this case bubble number 5. (LAB AND XRAY TESTING OF PATIENTS).

Expansion of Bubble #5 in "Hospital System"

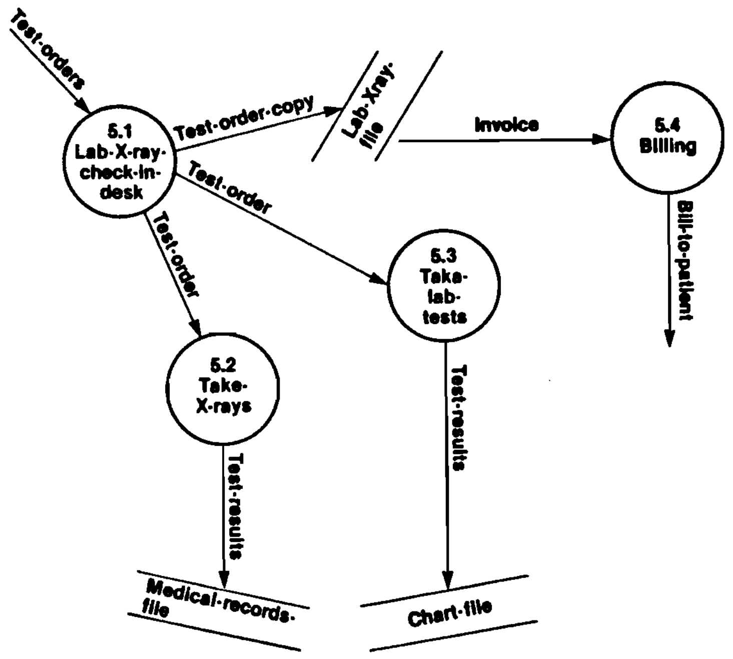


Diagram 5.0: Lab and X-Ray Testing of Data

Our "Test for Correctness" of this expanded DFD shows us that in the higher level DFD we had one input to process bubble #5 (TEST-ORDERS), and one output (TEST-RESULTS).

Here, however, we see two outputs! (TEST-RESULTS and BILL-TO-PATIENT).

Once again we immediately recognize an area of misunderstanding, and we return to talk to the participant in order to find out how the system really does work.

As we have seen, areas of misunderstanding can occur in data-flow path analysis. Also, there can be confusion about the exact definition of a particular data-flow file, or process bubble.

Structured Analysis contains a tool called the Data Dictionary, which attempts to eliminate ambiguity of definition.

DATA DICTIONARY

A SET OF DEFINITIONS FOR:

O DATA

O FILES

O PROCESS BUBBLES

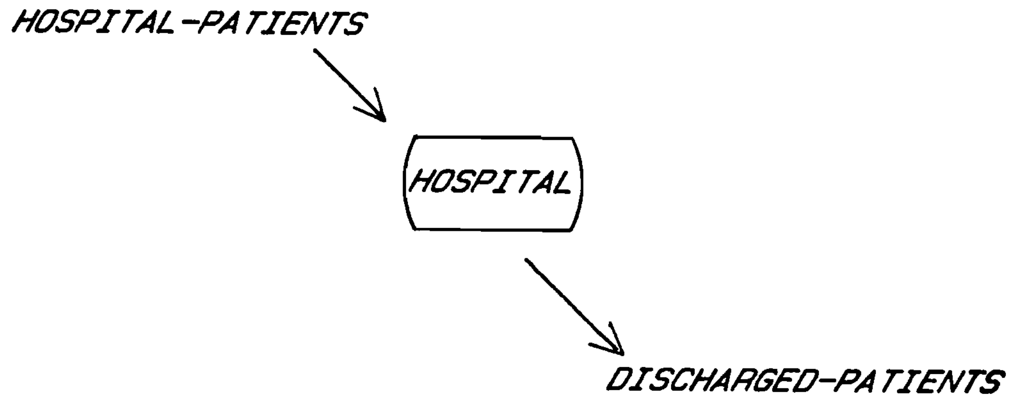
USED IN DFD

Here are some examples of Data-flow definitions in the Data Dictionary.

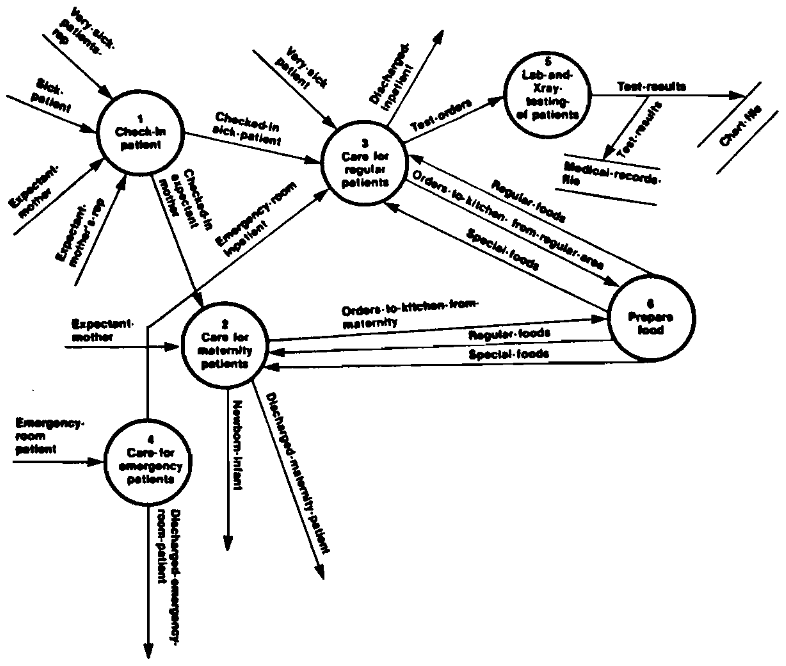
EXAMPLES OF DD ENTRIES FOR "HOSPITAL SYSTEM"

HOSPITAL-PATIENT (COMPOUND OR GROUP)	= SICK PATIENT OR EXPECTANT MOTHER OR EMERGENY-ROOM PATIENT OR VERY SICK PATIENT
DOCTORS ORDERS (ALIAS)	= TEST ORDERS
EMERGENCY-ROOM-PATIENT (PRIMITIVE DATA ELEMENT)	= "FLU" "AUTO-ACCIDENT" "HEART-PROBLEM"

HIGHEST LEVEL OF "HOSPITAL SYSTEM"



DFD of "Hospital System" Pass II



Expansion of Bubble #5 in "Hospital System"

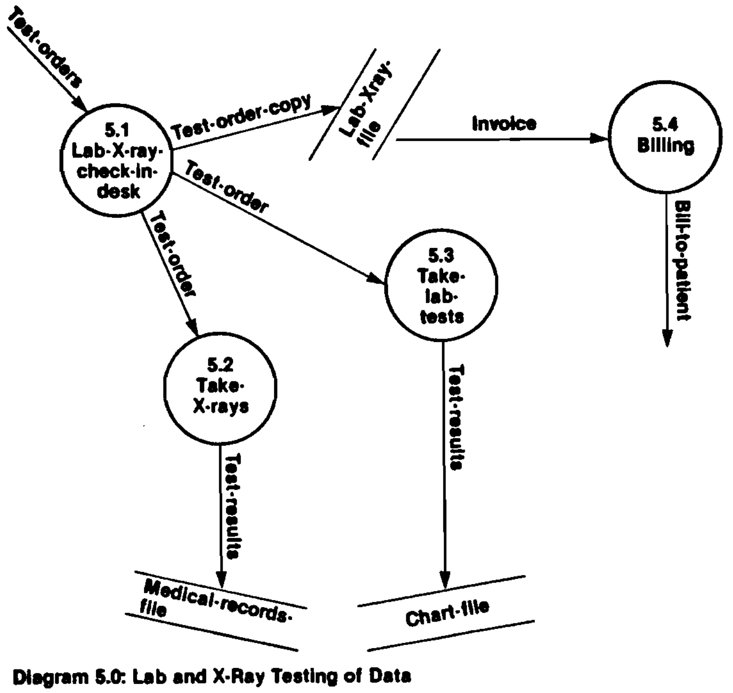
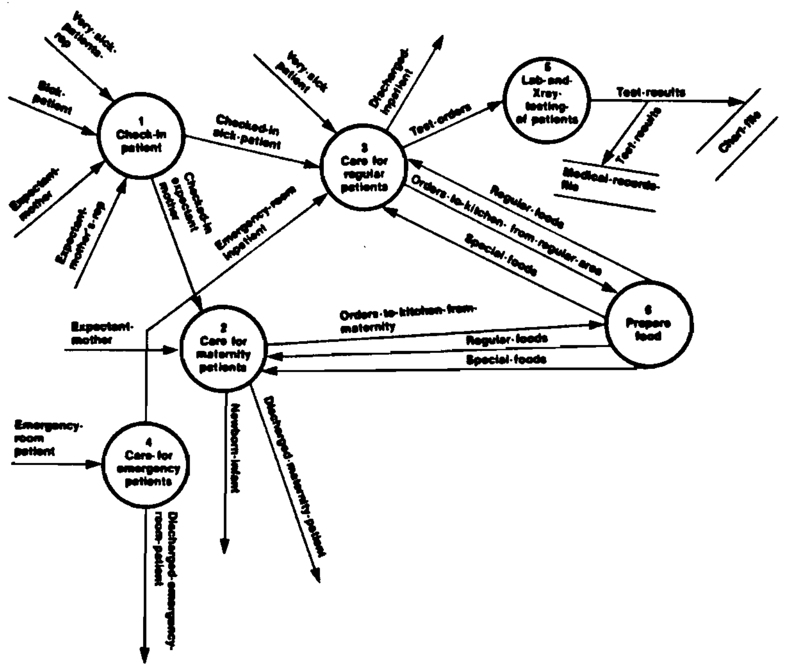


Diagram 5.0: Lab and X-Ray Testing of Data

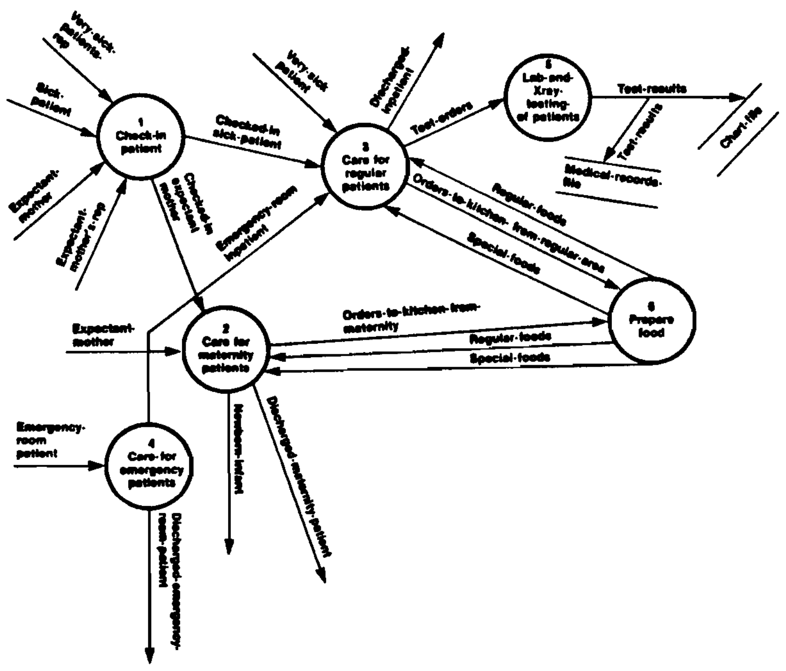
DFD of "Hospital System" Pass II



EXAMPLES OF DD ENTRIES FOR "HOSPITAL SYSTEM"

HOSPITAL-PATIENT (COMPOUND OR GROUP)	=	SICK PATIENT OR EXPECTANT MOTHER OR EMERGENY-ROOM PATIENT OR VERY SICK PATIENT
DOCTORS ORDERS (ALIAS)	=	TEST ORDERS
EMERGENCY-ROOM-PATIENT (PRIMITIVE DATA ELEMENT)	=	"FLU" "AUTO-ACCIDENT" "HEART-PROBLEM"

DFD of "Hospital System" Pass II



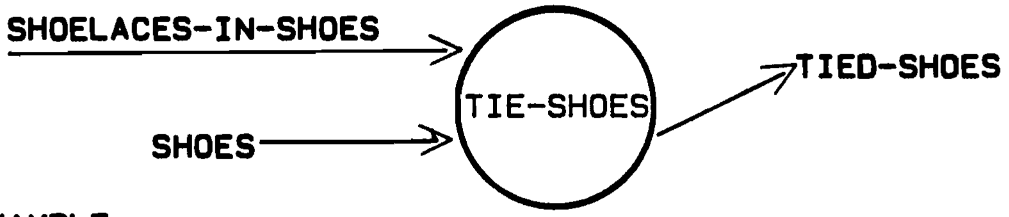
COMMENTARY

As we level our DFD for a system, each level of expansion shows more detail until we reach a level showing the primitive operations that act upon the data.

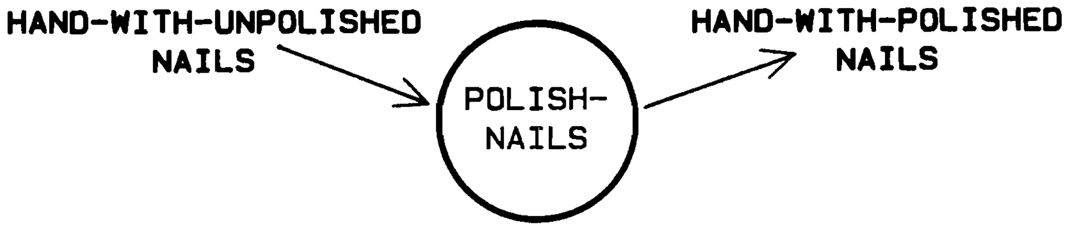
PRIMITIVE FUNCTIONS

PROCESS BUBBLES WHICH CAN NO LONGER BE EXPANDED REPRESENT PRIMITIVE FUNCTIONS WHICH ACT UPON THE DATA

EXAMPLE:



EXAMPLE:



Elements in the Data Dictionary which contain information about the process bubbles which are primitive functions are called Mini-Specs. Mini-Specs are written in Structured English.

Structured English

An Orthogonal Subset of English:

- Provides the minimum set of constructs needed to describe rules governing transformation of data flows for any functional primitive
- Provides one, and only one, possible way to describe rules governing transformation of data flows for any functional primitive

Policy for Preparing Foods

For each order-to-kitchen-from-regular area:

- For each special order:
  - —Collect foods needed to fill order
  - —Prepare foods
  - —Send special foods back to appropriate room
- For each regular order:
  - —Prepare foods
  - —Send regular foods back to appropriate room.

In summary, structured specification consists of:

- DFDs — pictorially shows relationship within the system
- Data Dictionary — defines the data acted upon by the system
- Minispecs — describes the primitive function which make up the system. These are written in Structured English.

Our Data Dictionary is a rigorous description/definition of all Data Flows, files and primitive functions which occur in the DFD which was derived from our Structured Analysis of a system.

Structured Analysis is a large topic. In preparing this paper, the most difficult task was in deciding what information to leave out.

I would suggest if you have further interest in the topic of Structured Analysis and feel the technique could be of use to you that you consult the following references:

- *Structured Analysis and System Specification* by Tom De Marco, foreword by P. J. Plauger
- *The Practical Guide to Structured Systems Design* by Meilir-Page-Jones, foreword by Ed Yourdon.