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).