Microcomputer programming skills

by C. WRANDLE BARTH

Goddard Space Flight Center—NASA
Greenbelt, Maryland

One of the most overwhelming aspects of the microcomputer revolution has been the speed with which hardware costs have plummeted. The software versus hardware costs ratio which prompted much of the interest in software engineering in recent years has grown dramatically as the parallel development of the microprocessors pushed down the cost of computer systems and their proliferation fueled software demand. This is increasing the need for skilled programmers for microsystems, part of which will be met by programmers currently working on big systems. In this paper we will examine some of the differences in skills and techniques that one can expect to encounter in the transition from programmer to microprogrammer. (The terms microprogram, microprogrammer, and microprogramming are frequently used in reference to the firmware or microcode of large machines. We will be using such terms only with reference to microprocessors.)

THE CONVERSION FROM PROGRAMMER TO MICROPROGRAMMER

Microprocessors are being used in a wide variety of applications. The programmer whose previous experience is with large batch systems may find that microprogramming for some of these applications requires a number of new skills. The hardware architecture, which high-level languages and operating systems have kept hidden from him on the big computers, may now be the primary thing with which he is working. Even the systems programmer who is accustomed to working at the level of assembly language and the machine architecture may find he must dig another level deeper, down to the logic diagrams and gates of the system. The ands and ors may have a familiar ring, but there are new things to unravel. If the application includes hardware development, signal timing diagrams will have to be read and understood to ensure the equipment being developed will work together; and testing and debugging now require oscilloscopes and probes as well as the more common traces and dumps.

These new skills are of particular importance when developing special-purpose systems that include microprocessors. When the hardware is being developed along with the software like this, one of the most important aspects of the work is the hardware/software tradeoff. Many of the functions to be developed can be accomplished in either hardware or software, or a combination of the two. The greater the designer's understanding of both areas, the more valuable he becomes in being able to create an optimal system.

If one is writing programs for micros in EDP, the changes in skills and techniques are much less pronounced. It will still be necessary to be more familiar with the hardware than would be required on large systems (although not to the degree needed in systems involving dedicated microprocessors). This is mostly prompted by the limited number of sophisticated tools currently available. Even though this is certain to become less of a problem as the necessary tools are produced, the EDP microprogrammer will have to bridge the gap for some time to come.

Tumbling costs are also making real-time EDP systems more viable. Programmers of such systems may have fewer hardware concerns than those programming dedicated microprocessors; but the interfacing is certainly much harder than for the batch-style EDP systems to which their experience may have been limited.

Many of the skills and techniques required in programming microprocessors are dependent on whether the micro is part of a dedicated, general-purpose batch, or general-purpose real-time system. These distinctions, in fact, are often overlooked in the presence of the much more obvious, but somewhat more superficial transition from maxi to micro. Although much of our discussion will apply to all areas of microprogramming, parts will be appropriate only to certain applications. We will attempt to distinguish among these areas in the topics that follow.

THE CONVERSION TO PROGRAMMING DEDICATED APPLICATIONS

With each new decrease in the cost of computers, new applications for their dedicated use have become economically justifiable. The maxis were limited to the really huge projects—defense, space and production control for large factories and utilities. The minis brought computer power to the smaller factories and research laboratories. And now the micros are making computers a part of everything from cash registers to video games. Those who are new to programming-dedicated microcomputers will not only need to learn techniques they never used in batch work, but they may
also find themselves using some techniques different from those of the traditional real-time programmer as well. The old ponderous real-time applications were frequently of a nature so large or critical that the only reasonable testing approach was simulation. With the smaller systems, it is more common to be able to test directly on a working prototype, using hardware test equipment instead of simulation.

As we have already noted, the knowledge of hardware/software tradeoffs is particularly important in projects in which both are being developed together. But such tradeoffs are not easily assessed. The cost of things accomplished in hardware recurs with every copy; the software costs occur only once. Thus there is a tendency to make the hardware "weak" for economic reasons. But this pushes more and more complexity into the software while trimming the hardware capability to a bare minimum. And when the application requires every last ounce of hardware power available, other things may have to be sacrificed. Such things as the use of high-level languages, various structured programming techniques, and the avoidance of coding tricks quickly fall prey to such an environment. Building systems that are friendly to the ultimate user requires resources that may be viewed as luxuries that cannot be afforded. Even high reliability or high programmer productivity can be lost when developing in a minimal hardware system.

THOSE PAINFULLY PINCHING SHOES

Developing programs in a minimal environment has always been burdensome. Dijkstra spoke of the earliest computers that were slow and whose memories were too small as "painfully pinching shoes." He said that the first programmers were pushed into coding tricks in the machine language and that they viewed programming primarily as the process of optimizing the efficiency of the computational process. Now, close to three decades later, we are faced with a brand new shoe of the very latest style, but one that is just as pinching as ever.

Many programmers view the microprocessor's relatively slow speed as immediately ruling out the use of high-level languages, regardless of the application. Optimization becomes glorified again, even optimization for its own sake. We know that optimization during coding is very error-prone. And with a dedicated computer, cycles saved on non-critical paths cannot even be made available to "other users." The use of programming tricks makes software even more unstable in these environments. (I should mention that I am not talking about programming idioms—such things as subtracting a register from itself to clear it—that become the universally acceptable way of accomplishing a task. It is the perversion of an operation code into a meaning unclear to other humans that often leads to errors.) In general, the smaller the excess hardware capacity over that minimally required, the harder our systems will be to program: the more the we will be pushed into poor programming practices; and the less reliable the end product will be.

But these are not the only aspects of machine smallness that the new microprogrammer faces. An easily overlooked difference is that less help is available from the operating system. In many cases, there is no operating system at all; just a bare bones machine. But even where some services are available, they are far removed from the facilities that were taken for granted on the mainframe. The programmer cannot count on a supervisor fixing a division by zero, or recovering from an input error. His code must do more checking and correcting for itself.

STRUCTURED PROGRAMMING ON MICROSYSTEMS

Structured programming has always required a healthy helping of common sense. The programming manager who decrees that "all programs shall be structured" is quick to discover that really terrible programs can be written without a sign of a goto and with no module exceeding 50 lines. Structured programming is not a blindly mechanical process, nor is it a panacea. The more demanding the external constraints, the more that sound judgment is required in applying the principles of structured programming; but also the more that the discipline will pay off.

Some people question whether structured programs are sufficiently efficient for use on micros. Of course, many of the techniques that have been loosely grouped under the name "structured programming" really have no bearing on efficiency at all. And while I've already indicated that I feel optimization is sometimes overemphasized, there is no doubt that certain constraints must be met, particularly in programming real-time applications (either dedicated or general-purpose). It is true that some structures which frequently turn up are inefficient. Redundant tests occur when using only standard control structures. Branch instructions whose targets are other branch instructions may be generated by high-level languages or assembly macros for some of the standard control structures. And numerous short subroutines can add substantial calling overhead to a program. Some of these problems can be overcome; others cannot.

Of major interest is the use of structured programming in high-level languages. As new languages are developed with the structured programming techniques in mind, we may find that better code is generated than would be for the current high-level languages. For example, optimizers can work better on structured code. Even our best current optimizers must abandon much of the optimization in program segments where a rat's nest of gotos prevents their deducing the flow of control. The optimizer for a structured programming language, however, could optimize every loop in the program since each begins with one of a small number of specific keywords. Furthermore, such optimizers could convert procedures that are called from only a single place in the program text into in-line code, thus cutting down on the calling overhead. But all of this is only what could be. What about what is? How should programs be written with the tools we have available now?

We know that most programs spend the majority of their time in a minority of their code. Even real-time programs are often time critical in relatively few places. These facts can allow us to take advantage of structured programming
techniques in our program development and then go back to post-optimize in those places where it is really critical. This also allows us to isolate our thinking about optimization from our thinking about the programming of the algorithm. Many of the techniques we use when optimizing are based on assumptions about the data of the program’s variables. If we optimize as we code, it is particularly easy to make false assumptions that get violated as we also optimize other parts of the program. Post-optimization is both easier and surer since all of the parts of the puzzle are present, allowing us to completely verify any assumptions we must make. We can attack the problem piecemeal, assuring each change is valid before making another. And most importantly, we can direct our efforts to those parts of the program that are most apt to be worthwhile.

Other software engineering techniques can provide efficiency payoffs. The use of functional cohesion and decoupling2 aid the complete replacement of algorithms with better ones. In a poorly modularized program, the code implementing an algorithm may be sprinkled about in such a way that it is impossible to change even when a better method is found. And replacing a poor algorithm with a better one can frequently yield a much higher optimization factor than can a ton of coding tricks.

The most important realization is that efficiency is a very relative thing. On a microprocessor dedicated to a single activity (whether used in a dedicated hardware system or in a general-purpose single-task system), wasting CPU time is of no concern if we are totally I/O bound and there is nothing else to do. Wasting memory is unimportant until it crosses a quantum boundary; and as chips become bigger and cheaper, the size of that quantum keeps increasing. Even when poor efficiency causes the system to be slower, it may be preferable. Efficiency considerations must be weighed against reliability considerations. Up to a point, a slow system that works is preferable to a fast one that fails (although a sufficiently slow system may be a failure by definition). A latent bug can be costly in any system, but particularly in a commercial device.

A NEW COMMUNITY OF USERS

One of the biggest changes that the microprogrammer faces is the new user community he serves. As a programmer of large systems, the “unsophisticated user” was a manager, a data entry clerk, a scientist, an engineer; with the micros it may well include a non-professional, an office clerk, a sales clerk, a houseperson, or a blue-collar worker. This is particularly true with programming for dedicated micros, somewhat less for real-time EDP, and least for batch EDP. However, even in this last case, the operator of such a system might well be the owner of a small business rather than a computer professional. The human interface of programs must take on a new aspect. These people will find words like “input,” “record,” or “string” strange or understand them differently and they are not accustomed to learning jargon. Moreover, they may not be just “down the hall” where they can ask you about some strange message they receive when trying to run your program. They are going to put commas in their numbers and type the letter “1” when they mean the digit “1.” And they expect “end” and “END” and “end” to all be the same—except when they want them to be different! But most important, they expect programs to work—and work right every time.

RELIABILITY

Consumers expect high reliability of the things they buy. No one expects to go to the store and find release 21.8 of a microwave oven! The closest things to “program fixes” that most consumers see are automobile company recalls. But if such recalls can hurt the images of the auto giants, think of what such bad public relations could do to a struggling company producing microcomputer-based systems.

And yet the overwhelming majority of programs are marketed while still sadly undertested. They are, in fact, so bad that it has become common to talk of buying program “maintenance.” But programs don’t break down; so what people are really buying is not maintenance at all, but a warranty. The programs are effectively guaranteed not to work as advertised; the “maintenance” assures that they will be fixed up when they inevitably fail. This viewpoint is well understood by people that have been dealing with computers every day, for they know that much software is quite bad. But the consumers are only starting to find out. This is not just limited to consumer-oriented systems, either. Customers have little sympathy for merchants whose real-time systems are down and prevent them from transacting business.

Is there a way to avoid bad software? There are some aids, for sure. Program certification (mathematical proof of correctness) offers the best hope, but is still far from practical for most programs. However, the more critical the application, the more certification may be justified, even if on a limited basis. Certainly programs that are a part of automotive systems are going to require higher reliability than they take over more crucial functions in those systems. For the time being, the best insurance for most non-critical software is probably the peer program walk-through. Such walk-throughs will frequently turn up bugs faster and better than testing methods will, and they have the additional advantage of being educational. One of the biggest problems that is overlooked by many is how will field upgrades be dealt with, particularly the inevitable fixes. For in spite of all their preventive measures, if the handling of the correction of a bug that slips through is mismanaged, a company’s whole reputation may go down the drain.

OLD LESSONS TO BE RELEARNED

There are a number of lessons that we have learned over the years that apply as much as ever to the micros or even more. To make sure we do not lose sight of them, we will review some here.

Use of high-level languages must be emphasized wherever
possible. Of course, there will be areas where the overhead
is just too great and we must turn to assembly languages.
But we must weigh this alternative carefully on a case-by-
case basis. We know we can turn out better software faster
using high-level languages. Even our "superprogrammers"
can (and we all like to think of ourselves as superprogram-
mers). But if we expect to keep up with the growing software
need, we must use the better tools whenever possible.

The same argument applies to using modularity. The gen-
eral-purpose routine seldom takes much longer to write than
a very special-purpose routine. And although it probably
won't be quite as efficient, it can save programming time
over and over if we can use it in future projects rather than
reinventing the wheel.

The biggest cost in changing hardware is changing soft-
ware. The big boys learned it when they were repeatedly
cursed with "downward" (backward?) compatibility. In
microcomputing hardware, six months is forever. The more
a given system is tied to a given architecture, the more that
system is going to become obsolete with the hardware. This
may be fine for dedicated use in a consumer product where
the time frame during which we plan on being in production
matches the expected availability of the component hard-
ware. But for general-purpose programs, software compat-
ibility will be the key to longevity. This adds another reason
in favor of using high-level languages. Even though different
compilers for the same high-level language may require some
changes in our programs, assemblers for different architec-
tures will require even more.

One of the biggest differences between good programs
and so-so programs is human engineering. This is all the
more important when we consider our new user community.
Programs should be written to be helpful and friendly to the
user. In many cases he will be communicating in an alien
environment and will not appreciate contorting himself to a
system with a thousand rules he can't remember. Sometimes
it requires a major increase in effort to build systems that
are well engineered; but frequently simple changes can pro-
vide great conveniences to the users as well as providing
additional selling points in a competitive market.

Patching is one programming technique that was falling
into disrepute on the larger machines and is now making a
reappearance. Patching object code rather than fixing the
source is sometimes required as a temporary measure when
testing micro software, particularly where the turnaround
time to recompile the program is long. The key to avoiding
efforts when patching is maintaining the discipline of keeping

track of all such changes and assuring that they get back
into the source. There is actually a greater problem than this
type of patching, and it is still seen regularly in all phases
of programming. That is the patching of source code con-
trary to the program's design. Sometimes such a patch is,
in fact, the only reasonable way to fix a design bug—partic-
ularly the last bug in a large design. If the time to correct
the error at the design level and then "recode" is too long,
then the source patch is just as reasonable as was the object
patch. And just as for the object patch, the documentation
should include complete information regarding the error for
future reference. But we should not be too quick to write
off such errors as unfixable at their true source. If any
further use is to be made of the program, it is all the more
desirable to correct the design error. And in well structured
programs, corrections in the design should be able to be
sifted down into the code fairly straightforwardly, allowing
the number of final modules that must actually be recoded
to be quite small.

CONCLUSIONS

The need for programmers for microcomputers is great
and will draw people from many disciplines. Professional
programmers for large machines can expect to have a big
head start if they move to micros and there will not be as
many differences ahead as they might think. Most of the
techniques that were applicable on the big machines are at
least worthy of consideration for the micros. although the
actual application may require some changes. The biggest
differences, however, will stem more from the differences
in applications than from differences in machine size. The
change to real-time programming is the most important of
these differences (particularly on dedicated microproces-
sors) and may entail dealing with hardware and machine
architecture to a greater depth. The other primary difference
is programming with a more computer-naive user in mind,
and dealing with his limitations and expectations.

REFERENCES

   15, No. 10, p. 860.
2. Yourdon, E., and L. L. Constantine, Structured Design, Yourdon Press,
   1975, pp. 116-186.