27
12-9-77
5 to NTIS

UCID-17557

Lawrence Livermore Laboratory

CDC 7600 LTSS PROGRAMMING STRATAGENS: PREPARING YOUR FIRST
PRODUCTION CODE FOR THE LIVERMORE TIMESHARING SYSTEM

K. W. Fong

August 15, 1977

This is an informal report intended primarily for internal or limited external distribution. The opinions and conclusions stated are those of the author and may or may not be those of the laboratory.

Prepared for U.S. Energy Research & Development Administration under contract No. W-7405-Eng-48.

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

THIS PAGE
WAS INTENTIONALLY
LEFT BLANK

AVAILABILITY

This document is available online as follows:

ELF RDS .717675:UCID:UCID17557 / 1 1

View the print file on the TMDS, or print it as follows:

TRIX AC / 1 1

.PRINT(<NIP UCID17557 BOX *ann identification*>)

.END

NOTICE
This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Department of Energy, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

CONTENTS

	Page
Abstract	1
Introduction	1
General Considerations	3
Portability	3
How Long Will the Code Last?	3
Debugging	4
I/O for Data and Results	4
I/O for Working Files	4
Overlaying	5
The CPU and Memory	6
Machine Cycles	6
STACKLIB	7
SSM	8
LCM	9
Pointered Arrays	11
Argument Lists	12
BEGINMAP, TICHEK	12
COMPASS	12
FTN	13
Input and Output	14
File Types	14
Single and Double Buffering	14
Triple Buffering	15
Overlapping I/O	15
BLIB76	16
Chained IOD's	16
Validating Disk Files	19
Debugging	20
Utility Routines	20
CHATR Options	21
Production Computing	22
Restart Dumps	22
System Crashes	23
ORDER Protocols	25
Multi-Controllee Jobs	28
Hardware Interrupts	29
BATCH, ORDER, and Your Controllee	29
Maintaining Sources and Binaries	32
Introduction	32
Managing Sources	32
Managing Binaries	33
COMPASS	34
COMPASS	35
Subroutine Linkages	35
Instruction Timing	35

CDC 7600 LTSS PROGRAMMING STRATAGEMS: PREPARING YOUR FIRST PRODUCTION
CODE FOR THE LIVERMORE TIMESHARING SYSTEM

ABSTRACT

This report deals with some techniques in applied programming using the Livermore Timesharing System (LTSS) on the CDC 7600 computers at the National Magnetic Fusion Energy Computer Center (NMFECC) and the Lawrence Livermore Laboratory Computer Center (LLLCC or Octopus network). This report is based on a document originally written specifically about the system as it is implemented at NMFECC but has been revised to accommodate differences between LLLCC and NMFECC implementations. Topics include: maintaining programs, debugging, recovering from system crashes, and using the central processing unit, memory, and input/output devices efficiently and economically. Routines that aid in these procedures are mentioned. The companion report, UCID-17556, *An LTSS Compendium*, discusses the hardware and operating system and should be read before reading this report.

INTRODUCTION

This document is the second of two reports (the first is UCID-17556) adapted from the series of on-line documents of the National Magnetic Fusion Energy Computer Center (NMFECC). The original documents were directed specifically to NMFECC users, but much of the material is relevant to the CDC 7600 computers on the Lawrence Livermore Laboratory's Octopus network. As a service to Octopus users, I have added some material pertaining only to Octopus so that my effort spent in preparing the original documents could be rounded out for the benefit of a larger user community. I hope you find these documents informative.

As you acquire experience in writing production codes for LTSS, you will develop your own methods and tools to suit your needs. At least, that is what we have observed of longtime LTSS users. For the most part, the NMFECC cannot supply you with and help you use other users' private utilities. We do, however, want you to get started in the right direction using the tools that are publicly available.

If you are a new NMFECC user, you probably have a large code that you would like to scale up to fill the 7600. This is bad for at least two reasons. One is that filling up memory without careful design of the I/O can be exceedingly wasteful of the machine's resources and can prevent other users from running their codes. Second, you are probably paying far more money to get your results than you need to spend. An existing code designed for some other system may also require considerable rewriting before it will run efficiently on LTSS. The fact that the 7600 may be ten times faster than some other machine does not mean

a code moved from the other machine will automatically be ten times faster. Nor does it mean the problem can be made ten times larger and finish in the same amount of time on the 7600. The conversion effort may be considerable, and you will have to decide whether a code should be converted or rewritten completely.

If you are a new Octopus user, you will start by writing small codes or by maintaining an existing large code. You probably will not have the problem that NMFECC users have, trying to convert a large code that already runs elsewhere. You should also keep in mind that the BATCH system described later is not used on the Octopus system. Also, CHATR is the NMFECC modified version of CHAT.

In the rest of this document I will recount the ins and out of LTSS programming, assuming that you are designing a new code and letting you decide how to impose a new design on an existing code. I will suggest some questions you must answer and indicate some answers proposed by pioneer users.

GENERAL CONSIDERATIONS

Portability

Do you really expect your code to run efficiently on more than one system? You should know by now that this is an unrealistic expectation. You will have to compromise somewhere. The first step is to modularize the I/O and system parts of your code as well as the computational parts. All dealings with LCM or I/O should be concentrated in a handful of subroutines so that system dependencies can be localized for easy modification to another system. You will have to avoid many of the nice features of LRLTRAN such as POINTER, BYTE, and ABSOLUTE statements. You cannot adjust field length dynamically. You cannot design your code as a set of controllees. You should use cliches (macros), conditional compilation (DIF statements), and PARAMETER statements. By setting parameter values and running your source through PRECOMP, you can extract different versions of your code suitable for different systems. Use only standard FORTRAN. Put only one character per word for Hollerith data. In FORMAT statements use the nH specification instead of single quotes, double quotes, or asterisks as delimiters. The number of things to consider is great, and it only begins with using standard FORTRAN.

How Long Will the Code Last?

A production code is not static. It will be run on many problems. Eventually you will come upon a problem you can almost handle, and you will want to modify the code. If your code is going to live that long, it will grow, so you must allow places in your design for new features to be added. Consider, for example, graphic output. If a code produces a couple dozen numbers, you can draw a graph manually. If it produces hundreds of numbers, you may wish to introduce plot routines. If the code is a long-running hydrodynamics calculation, you may want displays at a graphics terminal while the code is still running. You may want to produce a movie of a fluid calculation. If you have multiple species, you may need a color movie with each specie plotted in a different color. As you escalate your demands, you must consider whether your arrays are laid out in memory conveniently for plotting. Will you have space for graphics subroutines, or must you overlay? Other modifications to consider are: Did you allow for more general boundary conditions? Can your code handle more materials? Can your equation of state or tables of material properties be expanded? Will this two-dimensional code be the basis of a three-dimensional code? Is your input sufficiently flexible that you can enter information from a terminal as well as from card images?

Debugging

How will you debug your code? All input should be written down so you can later verify that the input was valid. As much as possible, check input for consistency. Write out intermediate results somewhere so you can see what happened if a code subsequently crashes. Write a private dayfile so you can tell where the code has been. Try to debug subroutines one at a time and not all at once. A critical quantity (such as a denominator) should be given a name and stored so that you can find its value from the dropfile with a debugging tool and see if its value is wrong (e.g., zero). The bigger a code is and the longer it runs, the greater is the chance it will crash. You should plan for this almost inevitable occurrence and produce enough information to diagnose the problem.

I/O for Data and Results

On input, the only problems are the ease of getting data in correctly and having the code verify that your boundary or initial conditions are reasonable. Output of results, however, can be expensive. Even when people plot pictures of the answers, they still generally insist on printing millions of numbers, ostensibly for reference or archival purposes. The problem here is that the BCD formatting routines use an enormous amount of time. The answer to producing a flood of floating-point numbers is to use vectorized formatting routines in BLIB76. ORDERLIB routines format one number at a time. The BLIB76 routines format whole arrays at once, using the parallelism available in the 7600. Also, you should get your numbers on microfiche, not paper.

I/O for Working Files

By working file we mean a scratch disk file needed only during execution as a temporary storage medium. At most, 16 (decimal) disk files can be connected to your program at any one time. In fact, 15 (decimal) are all that can be handled conveniently. Subtracting the input and output files and the controllee file if you need to read overlays, you have the number of working files that can be attached continuously. If this is not enough, you will have to open and close files as you compute and reassign IOC's as you go. If your code can be designed to do I/O to two or more files simultaneously, by all means use that many files. Keep your files as small as possible because other users need to use disk space too. Don't forget that it is most efficient to read and write multiples of the disk sector size, starting at disk addresses that are multiples of the disk sector size. Keep track of what information is where on disk rather

than using the two-argument BUFFER IN/OUT statements that have to read and write inter-record words between your data blocks. In effect, a disk is a memory that performs best when reading and writing whole sectors. Use it that way. Don't use it as a substitute for a tape, and don't use the binary read/write routines in ORDERLIB (i.e., FORTRAN unformatted reads and writes) until the new buffered binary routines are available.

Overlaying

Is your code so large that it requires overlaying? Do you expect to add features later that will make it too big to fit in SSM? You might consider dividing one large job into several smaller jobs. One job will leave its results in files for the next job to use. Often a job can be partitioned into initialization, calculation, and clean-up phases. The separate jobs can then be debugged separately. You can then write a controller (for example, using BCON) to execute the controllees in the proper sequence.

Another subtle point is that you may be able to divide a code into CPU- and I/O-bound parts that are logically independent. An example is solving a time-dependent system of partial differential equations at one time step while FR80 commands for plotting the solution at the previous time step are being written to tape. You write one controllee to continue your calculation and another to stage disk files to tape (for example, by being in turn a controller for one of the tape utilities). Then one controllee will initiate the other under a different suffix so that you have two jobs running concurrently (or, specifically, timesharing against each other). Since one job is CPU-bound and the other is I/O-bound, there is no serious contention between them for machine resources so long as they both fit into LCM simultaneously.

THE CPU AND MEMORY

Machine Cycles

The basic unit of time of the 7600 is the minor cycle, sometimes called simply cycle when there is no confusion with the memory cycle. A minor cycle is 27.5 nanoseconds (ns), and the speed of operations in the central processing unit is usually measured in minor cycles. The 7600 is theoretically capable of issuing a machine instruction every cycle -- a rate of about 36 million instructions per second. In practice, there are delays in reading instructions and operands from memory and other conflicts in the timing of operands arriving at the registers in the CPU that keep the rate of issue to 10 million to 18 million instructions per second for normal FORTRAN.

Floating-point additions and subtractions require 4 cycles to complete, and their results must be normalized at the cost of another 3 cycles. If you are lucky, the compiler may find some other operations to perform during the addition or subtraction; that is, independent instructions can be issued one right after another until you arrive at an instruction requiring the result of some previous instruction. The dependent instruction will be delayed until its operands or required functional unit are ready. The floating-point add unit is segmented and can accept another addition or subtraction on succeeding cycles. That is, it works like a pipeline: operands can go in on consecutive cycles even before the first result comes out. The floating-point multiply unit requires 5 cycles to produce a result and can accept more input every two cycles or more (i.e., not on consecutive cycles). The floating-point divide unit requires 20 cycles to produce a result; however, its iterative operation prevents the input of new operands any more often than every 18 cycles. From the relative times, it is clear you should avoid divides. Multiply by 0.5 rather than divide by 2.0. Integer addition and subtraction require 2 cycles. Integer multiply is 5 cycles. Integer divide requires a sequence of operations to convert integers to floating-point, do a floating-point divide, and convert the result back to integer.

The OPTIMIZE option is helpful for small, innermost DO loops. In LRLTRAN, DO loop indices, limits, and increments may be modified inside the loop. For the loop to work correctly, these quantities must be checked by reading from memory. If your DO loops are standard FORTRAN, the OPTIMIZE option tells the compiler it can keep these quantities in the CPU registers to reduce memory references. Furthermore, if your loop is small, the code generated by the compiler may fit entirely in the instruction stack so that, after the first time through the loop, no more instructions need to be read from memory for each iteration. Try to structure your code so the innermost loops are small.

IF statements tend to be expensive because they can branch to instructions not in the instruction stack, and there will be a delay until the desired instruction is read. If you know in advance which branch will occur more often

in a two-branch IF, place that branch immediately after the IF statement. This so-called fall-through branch is the fastest branch because the two-word look-ahead of the instruction stack will most likely have pre-fetched the instruction you need. For arithmetic IF statements, the zero branch is the fastest. For the arithmetic IF, the compiler tests for zero before it tests for positive and negative.

Double-precision is generally not needed on the 7600. Furthermore, it is not implemented fully in the hardware. Therefore, an expensive sequence of single-precision operations must be done to achieve a double-precision operation.

STACKLIB

The vector syntax and operations described in the CHATR manual are implemented by calls to subroutines in STACKLIB. You should design your code as much as possible around the idea of parallel or array processing. For example,

```
DO 10 I=1,N
10 A(I) = B(I) + C(I)
```

is a parallel operation because in principle all the additions could be done simultaneously and independently. You could rewrite the above statements in the vector syntax, asking for two vectors to be added together. Eventually a STACKLIB routine would be called to perform the operation. This routine is the best that can be written in assembly language. Thus, for sufficiently large N, the STACKLIB routine overcomes the cost of the subroutine call and will be faster than the DO loop even with OPTIMIZE. To enhance the portability of your code, the NMFECC suggests that you call the so-called dyadic and triadic vector subroutines in STACKLIB with normal FORTRAN subroutine calls rather than use the LRLTRAN vector syntax. If you call the routines yourself, you can use other memory increments besides 1. For example, the columns of a two-dimensional array are vectors with spacing 1, but the rows are vectors with spacing equal to the first dimension of the array. In a highly parallel code, the use of STACKLIB routines can cut the execution time in half. Recall that code produced by CHATR issues at the rate of 10 million to 18 million instructions per second and on the average is one-third efficient. The STACKLIB dyads and triads are handcrafted and will be more than 80% efficient. Thus, it is possible for a highly parallel code to run twice as fast using STACKLIB.

Another point to remember is that future super computers will have to incorporate many parallel or pipelined operations in their hardware in order to be significantly faster than the 7600. If your code is to run efficiently on such machines of the future, it will have to be structured so that you can easily take advantage of the vector hardware.

Finding inherent parallelism requires thought. We conclude this section with two examples. The first is Gaussian elimination where the fundamental operation is to multiply one row by a scalar and add it to another row. The multiplication of a row vector by a scalar is a vector operation if you conceive of the scalar as being broadcast into a vector of identical elements. Thus we are multiplying two vectors and adding a third. Since there are three operands (three vectors), this is a triadic operation. The STACKLIB routine that performs this operation is QMA4.

A second example is a matrix multiplying a vector:

$$c = Ab$$

One view of this operation is that the i -th element of c is the inner product of the i -th row of A with vector b . It is true that both ORDERLIB and STACKLIB have routines to perform inner products, but an inner product is not a true parallel computation. A better way is to say vector c is a linear combination of the columns of A . Using c as a temporary storage vector, calculate $c = b(1)*a(1)$, where $b(1)$ is the first element of b and $a(1)$ is the first column of A . Then $c = c + b(2)*a(2)$, and $c = c + b(3)*a(3)$, etc. Thus, the multiplication is carried out as a series of calls to QMA4. Either way, the same arithmetic operations are done, but the second way allows a sequencing that can be executed in less elapsed time. This same consideration may also influence your choice of algorithms. Orthogonal transformations by Householder's method requires inner products that cannot be resequenced, but orthogonal transformations by Givens' rotations can be vectorized.

SSM

Certain ways of using SSM are bad. To understand why, you must know how SSM works. SSM is arranged in 16 (decimal) banks with consecutive words residing in consecutive banks and wrapping around to the first bank every 16 words. A bank is busy for 110 ns when performing a read and 165 ns when performing a write. When a read instruction is issued, the addressing hardware initiates a request to the correct bank, and the bank becomes busy producing the required word. If the bank is not initially busy, 9 minor cycles elapse between the time the read instruction is executed and the time the word arrives in a CPU register. If the bank was busy from an earlier access, the arrival of the word must be delayed. The compiler tries to schedule reads as early as possible, so it can issue other instructions after issuing the read while it waits for the data from memory. References to SSM go through the storage-address stack (SAS). If busy banks cause two addresses to be held in the SAS, then instruction issue in the CPU is stopped. The current instruction may put one more request into the SAS for a maximum of three pending memory accesses. When the SAS becomes empty, instruction issue may resume. Note that the SAS is also used to read instruction words, so there can be a delay if the next instruction desired from

memory and the next data word are in the same bank. In writing from the CPU registers to memory, the write instruction finishes in one cycle as far as the CPU operand register is concerned and two cycles for the operand-address register. Memory then becomes busy, and the write is completed 9 cycles after the write instruction was issued. On a read, the memory bank becomes quiescent after 7 cycles even though the word it sent does not arrive at the CPU until the ninth cycle.

The net result is that SSM should hold the arrays that are accessed randomly. For sequential access, SSM is reasonably appropriate, provided you do not access the same memory bank in quick succession -- for example, writing B(I) and then reading A(I+1), where both are in the same bank. You can control this by putting arrays A and B in the same common block and dimensioning them appropriately or by putting in an extra dummy array to force the beginnings of A and B to the necessary separation.

Octopus 7600's are equipped with small core memory (SCM) instead of small semiconductor memory. If you are programming in a high-level language, you need not be concerned with the difference. SSM is slower than SCM but is more reliable. The timing differences matter primarily to codes written in assembly language, such as STACKLIB, which can be written for optimal performance on either but not both types of memory. STACKLIB was designed for SCM. Experiments have demonstrated that the degradation in performance on SSM is 5% to 8%. You should remember that SCM is arranged in 32 banks, so every 32nd word rather than every 16th word wraps around into the same bank. SCM requires 275 ns to read or to write. A read of SCM can be completed in 8 cycles, but the bank is still busy for a few cycles after the result is delivered. While SCM banks may be busy longer than SSM banks, the fact that there are twice as many banks and that reads complete in 8 cycles enables the overall performance of SCM to be a little better than that of SSM.

LCM

We need to delve a little deeper into how LCM works in order to understand how to use it effectively. LCM is arranged in eight banks. The smallest unit that an LCM bank can read or write is eight words. These units of eight words are called superwords or swords. Words 0 to 7 of LCM constitute a sword in bank 0; words 10 (octal) to 17 (octal) are a sword in bank 1. This arrangement wraps around so that words 100 (octal) to 107 (octal) are in bank 0. Each LCM bank contains a bank operand register (BOR). The BOR is a staging area for all reads from and all writes into LCM. A read out of LCM causes a sword to be transferred to the BOR. The desired word (in the case of a single read) or the desired words (in the case of a block copy) are then transmitted to the 60-bit CPU register (or to SSM). The LCM bank remembers which sword is in the BOR. Any subsequent read for a word that happens to be in the BOR can be made very quickly, because the word can be transmitted without initiating another bank memory cycle.

When you write in LRLTRAN, you can access LCM in four ways. One is a block copy from LCM to SSM. A second is a block copy from SSM to LCM. A third is a read of one word from LCM to one of the eight 60-bit registers (called X registers) in the CPU. A fourth is a write of one word from an X register to LCM. The individual reads or writes occur when you directly mention LCM common block variables or arrays in expressions or assignment statements. A further subtlety is that the compiler normally uses 18-bit registers (B registers) to calculate memory addresses, and an LCM address must be placed in an X register before you can read from or write to that address. Since the leftmost bit is the sign, address computations in B registers can compute an LCM address of at most 377777 (octal). To calculate LCM addresses greater than 377777 (octal), the compiler must use precious X registers, and the resulting code will not be so efficient. You must decide which LCM common blocks will extend beyond 377777 (octal) and force the compiler to use X registers (i.e., perform 19-bit addressing) by putting parentheses around those block names at the time they are declared to be in LCM.

We will now go into greater detail about how LCM works. If you cannot yet understand the details, try at least to remember the conclusions about what are the good and bad ways to use LCM. Consider first an individual read from LCM. A bank read/write cycle must begin if the required word is not in the BOR. Assuming the bank is not busy, it starts a read/write cycle to move the required sword to the BOR. Since the read of LCM is destructive, a write from the BOR back to the bank starts right after the BOR is filled. While the read/write cycle is completing, the specific word that was requested is sent from the BOR to the appropriate X register. If the bank is not busy to start with, it takes 17 machine cycles (minor cycles) for the word to arrive at the X-register destination after the read instruction is issued. Subsequent reads of words in the same sword require only 3 cycles each. A read/write cycle requires 63 minor cycles, of which the first 13 are used to read from the bank to the BOR and the remaining 50 are used to restore the bank. Notice that for sequential reading of memory, LCM is faster on the average than SSM -- 5 vs 9 minor cycles per word (5 vs 8 for SCM).

Writing an individual word to LCM is more complicated. The sword into which a new word is being written must be brought into the BOR. The new word replaces the old word within the sword, and the updated sword is written back to memory. If the bank is still trying to complete a previous read/write cycle, all further instruction issue will be stopped until that bank cycle completes -- assuming, of course, that the write wants to go into a sword other than the one that happens to be in the BOR already. Thus, reading from one sword and then writing to another sword where both swords are in the same bank is very bad. Writing sequentially to LCM is fairly efficient. The first two or three words to go into a sword will (if your code is fast enough in issuing writes) be in the BOR in time for the first write back to the LCM bank. That is, the first write from an X register triggers the read from the LCM bank to the BOR. While the sword is moving to the BOR, you have a chance to write in some more words to the same sword before the sword arrives. Theoretically, you can write four words to a sword before the write half of the cycle begins. Anything arriving after the write half of the cycle begins is held in the BOR (assuming it belongs in the sword that currently occupies the BOR), and a second LCM read/write cycle

starts when the first is finished, to write the late-arriving words to memory. Another effective way to use LCM is to read, modify, and rewrite words sequentially. Although this may cause the bank to go through three or four read/write cycles, the fact that the right sword is in the BOR allows the CPU to keep issuing instructions without any long LCM delays. The important thing to remember about using LCM is not to switch swords in a BOR too frequently.

Block copies are used to move contiguous blocks of words between SSM and LCM. The transfer rate, once the operation begins, is one word per minor cycle. A block copy requires that all LCM banks first become quiescent so that their usage can be synchronized for the block copy. This means that you should avoid an individual LCM read or write immediately before a block copy. Also, the storage-address stack (SAS) is used to direct data into or out of SSM, so it cannot be used to fetch instructions from SSM. Consequently, no instructions can be executed until the block copy is complete. A block copy causes a minimum of three LCM banks to become busy even if the block length is 1. Because block copies tend to make many banks busy, you should also avoid any reference to LCM within the first 60 or so minor cycles after a block copy.

Block copies serve at least three useful purposes. One is to move into SSM, arrays that must be accessed randomly because SSM supports random access better than LCM. Another is to move SSM data to and from an LCM buffer, because I/O can be done much more efficiently to your LCM area than to your SSM area. A third purpose is to use STACKLIB dyadic and triadic routines, which work only with SSM. The speed of these routines more than compensates for the cost of the block copies. Note also that BLOCKCOPY in ORDERLIB can be used to move blocks quickly from one part of SSM to another or from one part of LCM to another by using intermediate storage areas in LCM and SSM, respectively.

Pointered Arrays

In LRLTRAN there is a method of equivalencing variables dynamically by using POINTER declarations. For example, an array may have a pointer that is set to some location. Then a reference to the first element of the array is a reference to the location indicated by the pointer, and a reference to the second element is a reference to the next memory word; that is, the pointer holds the first word address of the array. A typical use of pointered arrays is to declare a large SSM or LCM array in an SSM or LCM common block. Working arrays are declared to be pointered, and then the location operator is used to set the pointers at appropriate addresses in the statically allocated arrays. The use of pointers adds another level of indirect memory referencing, because the pointer must be read first to determine its current value and then the address of the array element is calculated before the array element can be read. On the other hand, pointers are useful for partitioning large static arrays into smaller arrays and for controlling the origins of smaller arrays. This makes pointers useful for avoiding memory-bank conflicts, because now at execute time you can start the arrays in the banks with appropriate separation to avoid bank

conflicts. Inefficiency of indirect addressing is a problem only in a long sequence of code with many labels, because the pointer may have been reset just before a jump to a label. If pointered arrays are used primarily in DO loops with no internal statement labels or are passed as arguments to subroutines, they will not be more inefficient than normal, statically allocated arrays.

Argument Lists

Argument lists for functions and subroutines are considered long if they have six or more parameters to pass. The code generated by the compiler to pass long lists is different from the code for five or fewer arguments, and the long list requires more memory space and execution time per argument than the short list. Pass information through common blocks whenever it is practical.

BEGINMAP, TICHEK

Once your code is running, you should measure its performance to see that it is using the CPU as efficiently as you had intended. Subroutine TICHEK in ORDERLIB can be used to get an idea of how much time is used in each subroutine. BEGINMAP, also in ORDERLIB, along with the load map and long assembly listing of your code can be used to pinpoint small sections of code that use an inordinate amount of time. If a small piece of code uses a lot of time, see if it can be replaced with a STACKLIB routine. If no STACKLIB routine is suitable, you will have to consider using COMPASS or FTN.

COMPASS

COMPASS is both the name of an assembly language and the name of the assembler we recommend you use on LTSS. To use it effectively, you must be thoroughly familiar with the hardware and the timings of the various instructions of the 7600. Even then, it usually does not pay to write very large subroutines in assembly language. Consider COMPASS only for STACKLIB-style subroutines -- that is, operations that are simple and highly parallel.

FTN

If a subroutine uses a lot of CPU time but does not have local spots susceptible to STACKLIB or assembly-language subroutine replacement, then it must be compiled with a better compiler. The FTN compiler of Control Data Corporation produces code that generally runs in about two-thirds the time required by CHATR-compiled codes. Unfortunately, the code produced by FTN and the format of the relocatable binaries are incompatible with anything else on LTSS. As an alternative, you can compile by using FTN under the SLOPE subsystem on LTSS and get COMPASS card images back. These images are intended for assembly by the Control Data COMPASS assembler using a special set of FTN macros. By manually editing the assembly language output by FTN and redefining one or two macros, you can then assemble using LLL COMPASS, whose relocatable binary output is acceptable to the LTSS loader. This conversion is a manual process, and each subroutine tends to pose its own special problems in conversion, but it can be done. It works best on subroutines that do no input or output and have no external references. A project is currently (August 1977) under way to modify FTN to produce LOD-compatible binary cards. Ask your local compiler expert about progress.

INPUT AND OUTPUT

File Types

Choose a file format on which I/O can be performed efficiently. The binary tape-simulated format written and read by the binary read/write (i.e., unformatted) statements in FORTRAN is the least efficient format. It should be used only if your code needs to sacrifice performance for portability. (Efforts are currently under way to rewrite the binary I/O routines so that they will work more efficiently on the same format.) The monitor or squeeze-monitor format read and written by the two-argument BUFFER IN and BUFFER OUT statements is a little better but still inefficient. The best format is the absolute format where you specify exactly what you want to write to disk. You must then remember where to read from to get your data back. This kind of file can be written and read by the three-argument BUFFER IN and BUFFER OUT statements. As we have mentioned previously, this is the least you should accept if you intend for your I/O to be efficient. Since you have total control when you use the three-argument BUFFER IN/OUT, you can read the standard formats yourself and pick the data out from between pointer or inter-record words. Likewise, you can construct the image in memory of a standard format file with its pointer or inter-record words and write it to disk so that some other standard input routine can read it.

Single and Double Buffering

Single buffering on output means saving information in a buffer and then doing a disk write only when the buffer is full. On input it means reading a bufferful at a time and extracting logical records from the buffer as needed. When reading or writing past the end of the buffer, a disk read or write must be done before the reading or writing can resume. The solution to this delay is double buffering. On reading, you can continue reading the second buffer while the first is being refilled. On writing, you send the first buffer load to disk while continuing to fill the second buffer. The FORTRAN formatted read and write routines are double buffered. The FORTRAN unformatted read and write routines technically are also double buffered, but the buffer size used is so small that the routines for practical purposes are not buffered at all. An upcoming version of ORDERLIB, however, will have a new set of FORTRAN unformatted I/O routines where the read will be single buffered and the write double buffered. BUFFER IN and BUFFER OUT, contrary to their names, are not buffered at all; however, the three-argument BUFFER IN and BUFFER OUT allow you to implement your own buffering scheme. You must consider the space you will need for buffers, the amount of data to be transferred, the balance between I/O and CPU, and the amount of usage the code will receive. You want to increase

efficiency by making fewer but larger disk transfers, but you also want to keep the size of your code down. You want to use the optimal buffering scheme, but you don't want to spend months programming it. These are the matters you must resolve in designing I/O for your large working files.

Triple Buffering

Triple buffering is a computation technique often used in hydrodynamics calculations. Data is generally ordered on some grid or mesh, and you must sweep through the mesh, processing one row at a time. The data reside on disk primarily, because only a few rows' worth can fit in memory. Without buffering, you read a row, process it, write it out, and repeat the cycle on the next row. With triple buffering, you overlap I/O by defining three working areas or buffers in LCM. One area holds the preceding row and is undergoing a write to disk. A second area holds the current row on which calculations proceed simultaneously with the output. A third area is simultaneously being filled with data for the next row. Thus, input, output, and computing are all going on at the same time but in different areas of LCM. When the three processes are complete, calculation moves to the area where data was just read in, output begins on the row just processed, and input begins into the area that was just dumped to disk. This technique (also known as the three-ring circus) is a very efficient way to use a 7600. Sometimes two or three adjacent rows are needed in an iterative computation, so four or five buffers are needed. As in the three-buffer case, one area is writing data to disk, one area is reading data that will be needed next, and the rest are involved in a calculation.

Overlapping I/O

The goal of buffering is not to use more memory (which unfortunately buffering does) but to reduce the cost of I/O. Buffering can do this in two ways. One is to minimize disk accesses by packing information somewhere in an intermediate area (a buffer) until the area is used up, whereupon a read or write takes place. The other is to provide an intermediate area (a buffer), on which I/O may be done, that is separate from the area the CPU needs to access so that I/O and computing may occur simultaneously. In the latter case (called overlapping of I/O), you may not even think of the buffer as a buffer. It may look like an array that sometimes undergoes I/O. You try to get it filled before it is needed, and you try not to use it while it is being copied to disk. In this way you try to take advantage of the fact that CPU and up to four disk I/O operations on the MFE 7600 and the Octopus Z machine (three on Octopus R, S, and U machines) can be overlapped to reduce your cost.

BLIB76

The BLIB76 library contains many routines for doing I/O efficiently, although it also includes other routines. There are routines for reading and writing files that are single-buffered. There are also single-buffered routines for absolute files. Then there are conversion routines. If you insist on outputting millions of numbers (presumably to microfiche rather than paper), consider using subroutine MPAGE. This is probably ten times faster for large arrays of floating-point numbers than conventional formatted writes in FORTRAN. BLIB76 was written by applied programmers for use in production codes and contains many other good routines. Wherever BLIB76 and ORDERLIB coincide in capabilities, the BLIB76 routine is generally better and faster. In no case is it inferior.

Chained IOD's

LTSS allows more than one I/O request to be made at a time. Requests are fashioned into I/O descriptors (IOD's) that consist of four words. Usually, when you issue an IOD, you put a pointer in word 1 of your program to the location of the IOD, and then you perform an exchange jump. The IOD contains pointers to other IOD's if several are to be issued at the same time. IOD's may be contingent or simultaneous. Contingent IOD's are ordered, and the system will execute them in order. Simultaneous IOD's are not ordered and could be executed in any order convenient to the system, but currently they are also executed in the order in which they are chained. Chains of IOD's may contain both contingent and simultaneous IOD's, and the IOD's in a chain may perform I/O on the same file or on different files. There may be at most 32 (decimal) simultaneous IOD's, but a contingent chain may be arbitrarily long.

Contingent chains may be used when you must write many small records to tape. An IOD can write only a single tape record. Therefore, if many records are ready to be written, a contingent chain of IOD's should be issued. Some of the system tape utilities use this technique.

Much more important to the user are chains of contingent IOD's for reading disk files. We will illustrate their use in a typical situation where multiple reads are necessary from the same file. Suppose you are solving a time-dependent partial differential equation in a rectangular parallelepiped using finite differences and some alternating-direction or fractional-step method. Suppose further that you are using some implicit method that requires you to set up and solve linear systems of algebraic equations where the unknowns in the systems always lie in lines in the rectangular mesh. Let there be 80 mesh points in the x direction, 90 mesh points in the y direction, and 100 mesh points in the z direction. Assume that the solution is given at the boundaries for all time, so that only the interior points need to be calculated. Let l be

an index (subscript) in the x direction, J an index in the y direction, and K an index in the z direction. The x sweeps are done along lines parallel to the x axis. Sweeps for a given x-y plane are completed before incrementing K to move up to the next x-y plane. A sweep consists of using currently available function values along a line and its nearest eight neighboring lines to construct a linear algebraic system. Solution of the system gives an approximation of the solution on that line for the next time level. The new approximate solution is written over the previous approximation. After all 8624 x sweeps are done, 7644 sweeps in the y direction must be done. These also are completed one x-y plane at a time. Finally, 6864 sweeps along the z lines must be done. If the problem is nonlinear, it may be necessary to iterate this process three or four times to get a good approximation to the solution at the new time level.

We notice that there are 720,000 mesh points, so it is not possible to put all 720,000 function values in LCM simultaneously. The unknowns must be suitably ordered and stored on disk. (In the rest of this section we assume that 819 or 817 disks are being used, but the same reasoning applies to 844 disks.) The ordering we choose is for adjacent elements on x lines to be contiguous, adjacent x lines to be concatenated, and adjacent x-y planes to be concatenated. The concatenation is not strict, in that there may be gaps on the disk in suitable places, but the ordering is always preserved. Let us assume that we have an initial approximation and are cycling through x, y, and z sweeps to improve it. Suppose that the x-y planes consisting of 7200 elements each are spaced so that they each start on a sector boundary. This means that they use 15 (decimal) sectors each and waste 480 words in every 15th sector. To perform y sweeps in an x-y plane, you should triple buffer. At level K in z, you need planes K-1, K, and K+1 for constructing difference equations. Level K-2 is being buffered to disk, and level K+2 is on its way in for future use. Since whole planes are moving at a time and since the planes are laid out as 7200 contiguous elements on disk, only one IOD is needed at any one time. Notice further that no useful information is in the last 480 words of the 15th sector. This means that the IOD may specify that the unreferenced portion of the last sector need not be preserved. This cuts the cost of the disk write.

Having finished the y sweeps and written everything to disk, we next wish to do z sweeps. The sequence is to process z lines in a given x-z plane and then move on to the next x-z plane at the next J level. To get an x-z plane into LCM, we must read all the x rows in that plane. However, these rows are spaced 15 sectors apart on the disk, so we must issue 100 IOD's to get a plane in. Clearly, chained, contingent IOD's are needed. You could, of course, make 100 exchange jumps, each issuing a single IOD, but this would waste CPU time. In performing the z sweeps, you again triple buffer in the sense of reading and writing x-z planes simultaneously while using three others in a computation. When an x-z plane is finished and ready to go back to disk, we face a problem: it is easy to read x lines from anywhere on the disk, but it is hard to put them back where they came from. The reason is that they were pulled out of the middles of sectors, but they cannot be put back without preserving the unreferenced portions of the sectors they came from. Preserving unreferenced portions of sectors, you will recall, is expensive because it requires that the sectors be read into the PPU. The solution is to use another disk file for

writing the x-z planes. The planes presumably are laid out in an orderly way in LCM, so they can be moved out to disk with one IOD per plane. As in the case of x-y planes, it is convenient to start each x-z plane on a sector boundary, so preservation of the first sector is unnecessary.

When the z sweeps are finished, the x sweeps are done again. It is now necessary to recover x-y planes. This means that 90 IOD's must be issued to pull the appropriate x lines (which are 16 sectors apart on disk) back together in LCM. The x sweeps write x-y planes that are also used by y sweeps, so nothing fancy is required when completing x and going to the y direction.

To summarize the example, we have seen that each of three kinds of sweeps requires that planes be read and planes be written. Planes are always written with single IOD's. The x sweep reads with chained IOD's because the z sweep left the rows in the wrong permutation. The y sweep uses x-y planes as did the x sweep, so it can read with single IOD's. The z sweep needs x-z planes so it reads with chained IOD's the x-y planes written by the y sweep. At least two disk files are necessary. For ease of restarting after crashes, it would be desirable to use three files. The y sweeps should not write x-y planes into the file from which they were read. If you use three files and there is a crash during a y sweep, it is necessary to back up only to the beginning of the y sweeps, not to the beginning of the x sweeps.

There are a few other interesting things to know about IOD's. IOD's are always scheduled by the operating system on a first-come-first-served basis. This is true even for chains of simultaneous IOD's; that is, the IOD's are issued in the order in which they are chained. This raises questions of whether a chain of IOD's can read every n -th word in a cylinder where n is a multiple of the sector size (or more generally, reading m words starting at every n -th word but not reading past a sector boundary). The answer is that the PPU is fast enough to read every other sector without losing a revolution. Therefore, such a chain of, for example, 50 IOD's (with n = sector size) should be ordered to read 25 even sectors and then 25 odd sectors to pick up all the data in two revolutions. If they were arranged in the obvious monotone increasing order, over 51 revolutions would be required. If the same question is posed about writing something to each of 50 consecutive sectors in a cylinder, the answer is that the PPU is fast enough to write to every third sector, provided it does not have to preserve the unreferenced portions of the sectors. Thus, in the best case, three revolutions are needed to write segments into 50 consecutive sectors. (You would generally be better off to construct the image of the 50 sectors in LCM and write it all out with a single IOD.)

How do you tell where in the cylinder your file starts? You must read the appropriate IOC by issuing system call 0704. The IOC includes the absolute disk word address where the file starts. It will always be a multiple of 512 words. Furthermore, all files start with an ID block of 1000 (octal) words not generally available to the user. Thus, a file of length 1000 (octal) is really 2000 (octal) words long. Your word 0 is really 1000 (octal) words from the absolute beginning of the physical file. There are 102,400 (decimal) words in

an 819 cylinder, so you can tell where a file starts modulo 102,400. [*]
Exactly the same reasoning applies to 844 disks, except there are only 29,184 words per cylinder. The 817 has a cylinder spread over two spindles of 81,920 words.

Validating Disk Files

It is possible for a disk sector to become flawed. When the system detects a bad sector, it adds that sector to its flaw table so that no one in the future will be given that sector as part of his disk file. Unfortunately, some poor user is usually victimized by a bad sector before the system learns the sector is bad. If you are creating a disk file for an important run, you should write something into every word of the file and read it all back. Bad sectors are usually found while reading rather than writing. Flawed sectors as well as other problems are returned with the *f* field set to 2 in the IOD. If you get a disk-parity error, destroy the file and create it again. If the sector truly is bad, it will have been added to the system's disk flaw table, and you won't get it again when you recreate the file.

[*] m modulo n is the number between 0 and $n-1$ that results from subtracting from m the largest multiple of n that is less than or equal to m ; e.g., 38 modulo 11 is 5, and 33 modulo 11 is 0.

DEBUGGING

Utility Routines

It is unrealistic to expect a large code to be error free. Presumably, part of your output is a log or record of what your code did. If this record is not enough to explain wrong answers or a job abort, you will have to save the controllee's dropfile and look at it. If you need to dump most of your arrays and variables, use VDUMP or DOD. The latter is more useful if you have a symbol table for the controllee. ORDER uses DOD when you use the *DUMP control card. Core dumps are now discouraged, however, because there are more convenient tools to use if you need to see only a few words in your controllee. DBUG may be used to look at variables symbolically. It finds from a symbol table the location of a variable in the controllee and can then print its value at your terminal. If you do not have a symbol table (shame on you) for your controllee or if you need to examine some other kind of disk file (e.g., seeing what was really written in your scratch files), you will need utility routine EDIT or VDUMP.

CTRL is a dynamic debugger that acts as a controller for the controllee to be debugged. Its essential capability is to put breakpoints into your controllee. A breakpoint is a location where the code will stop executing if it tries to execute the instruction at the breakpoint. The purpose of breakpoints is to allow you to examine the controllee at a time prior to its failure so you can see how it starts to go wrong. CTRL cannot be used conveniently on overlay codes. There is also a dynamic mode for DBUG in which it controls CTRL, which in turn controls your program. The interface is unreliable, however, and the utilities may crash. CTRL does not require a symbol table. It depends on you or DBUG to give it octal addresses for setting breakpoints. Likewise, when CTRL is used alone, you must tell it the octal addresses of memory locations you want dumped.

A much better tool is DBCTRL, which, though still under development, is proving to be more useful and reliable than DBUG and CTRL. It requires a symbol table along with your controllee. In static mode (i.e., looking at your controllee after it has died), you can ask for the final values of variables by name. The commands to DBCTRL are less ambiguous than DBUG commands. In dynamic mode DBCTRL has many capabilities heretofore unavailable. It can set breakpoints in overlays easily. You can keep a copy of the dropfile at a breakpoint so that, if your code crashes before it gets to the next breakpoint, you can back up and try again. DBCTRL is a public file on the MFE 7600 and may be read from photostore on the Octopus network under directory chain

283015:MFE PUB:CONTROLLEE:CURRENT:DBCTRL

Another 7600 debugger being developed is DDT. It is not publicly available at this time, August 1977.

CHATR Options

Two debugging options in CHATR (and CHAT) may be helpful to you. They are described in the manual, so we will describe them here only briefly. CHECK can be used to print the name and value of a variable whenever you store into it. This may help you find how wrong values get into variables. The LINETAG option puts out extra labels in the symbolic cards for building the symbol table. The fictitious labels are based on the line numbers of the code. Thus, the location of the pseudo-label goes into the symbol table. With DBCTRL you can find the corresponding location in your controllee and set a breakpoint there. This is helpful if you must set a breakpoint where there is no natural label nearby. Also if your controllee crashes and leaves the program counter intact, you can pinpoint more closely the line of code you were executing when the error occurred.

PRODUCTION COMPUTING

Restart Dumps

There are two situations in which jobs need to be restarted. One is where the job is so long that it is not allowed to run to completion. It runs a few hours each night until it is done. The other is a system crash where all user jobs are interrupted. Inasmuch as all production computing is done under the control of the ORDER subsystem (on the MFE 7600, at least), we will discuss restarting only in terms of the features available under ORDER.

The simplest type of restart is to save the dropfile and all working files and restart by executing the dropfile. If you are using tapes (which you shouldn't be doing), this won't work, because the tape will not be positioned on the tape drive where your dropfile thinks it ought to be. Even if you are not using tapes, it may be inappropriate to keep all files and simply re-execute the dropfile. For example, an output file may have information you would like to see. Therefore, you would like to send an output file to a device and create a new file for further output when the job is restarted.

In order to allow for a more graceful restart, your job must know when it is about to be interrupted so it may clean up files and make a graceful exit. Essentially, there are two ways your job can get sufficient warning. One is to use the *TIME control card in ORDER. This is a card you should be using. The other way is for the operator to send a SWON message to ORDER when he wants you to get off the machine. Either way, sense switch 1 is turned on or an appropriate message is sent to your job by ORDER. You must check continually for sense switch 1 being turned on and prepare to leave the machine when it comes on. Two ORDERLIB routines may be of use to you in this process. One is OFFMON, which clears buffers, gives away certain files, and sets the program counter for a subsequent restart. The other is EXIT (with one argument) for exiting but leaving the program counter set for a restart. Your job notifies the system that it is through executing, but that the dropfile should not be destroyed. In a subsequent task in the same ORDER run, you save the dropfile for re-use. On another run, you read the dropfile and working files into your file space and restart with a *XEQ card. If your controllee was originally loaded under control of ORDER (i.e., you used a *CONTROLLEE card but not *ZEROLOAD card), then you must use a *ZEROLOAD card when restarting the dropfile.

An ORDER-loaded controllee is one that has the ORDER resident package as code block at level 1. The user's main program is in a code block at level 2. You get the resident package by allowing ORDER to tell the loader how to load your controllee. ORDER then forces the resident package in ahead of your code. This package has error-handling routines and also sends the dropfile name back to ORDER. When ORDER executes a controllee, it first looks for the presence of the resident package and expects a certain interaction to occur if the package

is there. When you restart an ORDER-loaded dropfile, the resident package is there, but the initial interaction occurred on a previous run; therefore, the *ZEROLOAD card tells ORDER not to expect any messages characteristic of an initial execution. If you used a *ZEROLOAD card when you created the controllee, then your controllee does not have the resident package and is considered not to be an ORDER-loaded controllee. In this case the *ZEROLOAD card is not needed when you restart the dropfile on a subsequent run.

System Crashes

The hardware and operating system are not infallible. Memory-parity errors and other problems may necessitate a dead start of the operating system. Serious problems can happen to your job in this situation. Tapes are removed from tape drives on a dead start. If you were writing a tape (which you should not be doing) or if you were running a utility routine that was writing a tape, you will need some error-recovery procedure to verify that the tape was correctly written. The system may also have lost its copy of the file index in LCM and had to back up to a copy on disk that is a couple of minutes old. If there is a dead start between the time you create a file and the time you write to it, your dropfile believes the file exists, but the old file index says it doesn't exist. The system will then abort your job for writing to a non-existent file. Again, an error-recovery procedure must be used after a dead start to be sure all necessary files exist. However, it is not enough for requisite files merely to exist. If a scratch file is being used for reading and writing, it is possible to lose synchronization. For example, suppose you read a file, rewind it, and write something new on it, and then there is a system crash. Under production, it is quite possible that your job has been in memory for over a minute, and that the dropfile on disk represents the state of your program before it did the read and write. On a crash, your core image could be lost, forcing a restart with whatever dropfile is left on disk. This dropfile, however, will read your working file as though it had not yet been rewritten, thereby getting the wrong information. In this case, you may not be able to recover if you have not allowed for some method to reconstruct your working file. A partial solution is to write some sort of identification to disk along with your data, so that you can verify that the right set of data is on disk on a subsequent read. A better idea is to write to another disk file and not overwrite the one you've just read until enough information has been written elsewhere that you no longer need the first file.

Thus far we have mentioned error-recovery procedures without describing typical implementations. Under production, the operator (on Octopus) or BATCH (on MFE) will restart ORDER's dropfile with a LINEFEED-D message. This is a special message to ORDER to inform it that a dead start has occurred and that it will have to restart your job. If your controllee is not ORDER loaded, ORDER will start your original controllee, not the dropfile, because ORDER generally will not know the name of the dropfile. Also, ORDER will restart your controllee without a message even though a *XEQMES card is present. A *RSTMES

card is needed to specify a message upon restart. For ORDER-loaded controllees, ORDER knows the dropfile name and will restart it without a message unless a *RSTMES is used. A *RSTNAM card can also be used if you want ORDER to restart something other than what it would normally restart. Thus, one way to implement a recovery procedure is to make it part of your code. You get a message from controller (with a system call) and see whether the message indicates a normal start or dead start. If there is a dead start, you first verify the existence and integrity of files before starting to compute. Your code should be arranged to create dump files for restarting every half hour. That is, in case of a dead start, the dump files have enough information to restore working files and arrays and variables in memory so that you may resume from the most recent dump and not lose more than a half hour of machine time.

There is another way to use the above control cards. You may have a special controllee to do recovery, which you specify with the *RSTNAM card. This puts the recovery code in a separate controllee (which must exist after the dead start) and keeps the main controllee smaller. After restoring files, you call ORDERLIB routine CBRANCH to switch the normal exit branch to a task that will execute your main controllee again.

In addition to the control cards, there are subroutines in ORDERLIB you may call to accomplish the same or additional functions. SDRPNAM tells ORDER what file to restart. This is useful if your controllee drops to a random dropfile name that cannot be known at the time the control cards are set up. It is also useful if you drop a second time to yet another dropfile name. SRSTMES deposits with ORDER the message you want to be restarted with after a dead start. If there are several stages in your cycle of computation from which you can recover, you want the message to tell you how far you got before the crash. GRSTMES is used to get from ORDER the restart message (even if there has not been a restart) and a flag indicating whether a dead start has occurred since the last call to GRSTMES. WARNING: Be careful about putting calls to GRSTMES in short loops. It asks ORDER if there has been a restart, but, in doing so, it forces your controllee to drop to disk. Thus, it costs you a voluntary load and dump that will be added to your I/O charge.

The correct use of control cards and the ORDERLIB routines is the basis for designing a restart procedure for a production code. The first step is to design your code to have some sort of loop structure. At one (or more) point(s) in the loop you test for indications that a dead start has occurred or that you are about to run out of time. To set sense switch 1 to indicate that time is nearly up, you should use the option on the *TIME card that implements sense switch 1 solely by setting a bit in the 24th (decimal) word of GOBCOM. This is option l=0. Option l=1 should be avoided because the message may get lost in a dead start. Option l=2 should be avoided because the IF(SENSESWITCH,1) statement will first toggle sense switch 1 if there is a SW1. message waiting for your controllee. If the switch was already on (meaning you should terminate your job soon), it will be turned off, and the branch for the off position will be taken. Option l=3 should be avoided because you will then receive no indication that you should terminate, and ORDER will abort your job when your time is used up. The IF(SENSESWITCH,1) test is safe for option l=0.

Alternatively, you can declare common block GOBCOM to be length 24 (decimal) or 27 (octal) and look at bit 6 in the 24th (decimal) word. (The bits are numbered from 0 to 59, right to left.) This bit stands for sense switch 1. A value of 1 means on; a value of 0 means off. If this bit was turned on by ORDER before a system crash, it will still be on after the restart. It will not be lost as a SW1. message would be. After checking the sense switch, you should check for a dead start by calling GRSTMES and checking the value of the flag. All the routines described in the ORDER JOB CONTROL section of the ORDERLIB manual send messages to ORDER and receive messages from ORDER. If you are expecting any messages from controller, you should pick up the message from the operating system (using a get message or symbols from controller system call) before calling any of these routines, because ORDER will need to use the system's buffer to send a message back to these routines. The system buffer for conveying messages from controller to controllee holds only the latest message. Messages not picked up by the controllee will be lost when the controller sends another message. In particular, the SW1. message, unless picked up first, will be wiped out by a call to these ORDERLIB routines. If the flag returned by GRSTMES indicates a dead start has occurred since the last call to GRSTMES, then you branch to a part of your code to check the status of all your files. To see if a file exists, close it and open it. No error will occur on the close, even if the file to which you were connected disappeared during the dead start. On trying to reopen it, you will get an error indicating a non-existent file. After the file is recreated or reopened, you can restore its contents. Because a call to GRSTMES causes a voluntary load and dump, it should be used only when you can afford these extra I/O charges. For a controllee that uses all of LCM, you should use the *RSTNAM and *RSTMES control cards to execute something other than your dropfile after a crash. In particular, you might want to start a fresh copy of your controllee but with an alternate execute message.

ORDER Protocols

If your controllee is written in BCON rather than LRLTRAN, the ORDERLIB routines for talking to ORDER are not available to you. You therefore may need to know the protocols and formats of messages. ORDER has its input bypass closed and its output bypass open when it initializes your controllee. You must issue a system call to see if +ORDER is your controller. If it is, you must issue a system call to close the controller output bypass. When ORDER replies to your controllee, it will send a message and restart your controllee with ORDER's output bypass again open. The ORDERLIB routines use a local 12 (decimal)-word array named BETA. Remember that calling any of these routines will incur a voluntary load and dump charge. Let CTLO be 400217 (octal).

1. CBRANCH

To ORDER: BETA(1) = (5H CHNG).UN.CTLO

BETA(2) = IFLAG
 BETA(3) = IBUFF(1) Note: The names in BETA(3)
 BETA(4) = IBUFF(2) to BETA(8) must be
 BETA(5) = IBUFF(3) right-justified.
 BETA(6) = IBUFF(4)
 BETA(7) = IBUFF(5)
 BETA(8) = IBUFF(6)
 BETA(9) = EOM
 BETA(10) = BETA(11) = BETA(12) = 0

From ORDER: BETA(1) = 2ROK

2. GBRANCH

To ORDER: BETA(1) = (5H GBR).UN.CTLO

From ORDER: IBUFF(1) = BETA(1) normal branch name
 IBUFF(2) = BETA(2) software error branch name
 IBUFF(3) = BETA(3) hardware error branch name
 IBUFF(4) = BETA(4) ORDER abort branch name
 IBUFF(5) = BETA(5) interrupt branch name
 IBUFF(6) = BETA(6) zero repetition branch name

3. SBRMES

To ORDER: BETA(1) = (5H SEND).UN.CTLO
 BETA(2) = NUM + 1
 BETA(3) to BETA(n) = message, n .LE. 10
 BETA(n+1) to BETA(12) = 0

From ORDER: BETA(1) = 2ROK

4. GBRMES

To ORDER: BETA(1) = (5H GET).UN.CTLO

From ORDER: IBUFF(I) = BETA(I+1) for I=1,...,NUM

5. SETREP

To ORDER: BETA(1) = (5H SREP).UN.CTLO
 BETA(2) = NUM

From ORDER: BETA(1) = 2ROK

6. GETREP

To ORDER: BETA(1) = 5(H GREP).UN.CTLO
BETA(2) = BETA(3) = 0 if only one argument in call
BETA(2) = IFLAG
BETA(3) = ITASK

From ORDER: IREP = BETA(1)

7. RSTMODE

To ORDER: BETA(1) = (5H MODE).UN.CTLO
BETA(2) = NEW

From ORDER: OLD = BETA(1)

8. SRSTMES

To ORDER: BETA(1) = (5H SRST).UN.CTLO
BETA(2) to BETA(11) = message with zero fill from right
BETA(12) = 0

From ORDER: BETA(1) = 2ROK

9. GRSTMES

To ORDER: BETA(1) = (5H GRST).UN.CTLO

From ORDER: MESS(I) = BETA(I) for I=1,...,10
IFLAG = BETA(11)

10. SDRPNAM

To ORDER: BETA(1) = (5H SDRP).UN.CTLO
BETA(2) = NAME

From ORDER: BETA(1) = 2ROK

11. GDRPNAM

To ORDER: BETA(1) = (5H GDRP).UN.CTLO

From ORDER: NAME = BETA(1)

12. SORDMES

To ORDER: BETA(1) = CTLO

BETA(2) to BETA(n) = message, n .LE. 12

BETA(n+1) to BETA(12) = 0, n.LE. 12

From ORDER: BETA(1) = 2ROK

13. GORDBOX

To ORDER: BETA(1) = (5H GBOX).UN.CTLO

From ORDER: IBUFF(1) = BETA(1)

IBUFF(2) = BETA(2)

IBUFF(3) = BETA(3)

14. GORDLIB

To ORDER: BETA(1) = (5HGPLIB).UN.CTLO

From ORDER: NAME = BETA(1)

Multi-Controllee Jobs

As mentioned previously, it may be advantageous to design a single job as several controlleees that are run in succession or in a cycle. Each controllee hands information to its successor by leaving disk files of data. This means the job is automatically well structured for restart dumps and dead starts. Then all you need is a controller to run the controlleees in the right order, and your controller is run as a controllee of ORDER. In this case, it may be convenient to write your controller in BCON and to put error recovery and restart logic in the controller. One of the multiple controlleees can be devoted to restarting. It can read the I/O connectors of the controllee that should be executing, to see what files it is connected to, and then check the file index to see if the files still exist. When it has restored all files, it indicates

to the controller that execution of the controllee that died in the system crash may begin again.

Hardware Interrupts

For errors such as overflow or indefinite operands, which are trapped by the hardware, your controllee is usually aborted; however, if you think you can write a section of code to recover from these and certain other hardware-detected errors, the method for regaining control of your program is described in the SYSCALLS document (or LTSS Chapter 10). Notice that an underflow is not normally considered to be an error, but you can make it an error if you so desire.

Using this recovery procedure requires setting special information in word 174 (octal) of the minus words. This can be done with utility routine EDIT after your controllee is loaded, or it can be done at load time by DATA-loading a variable that is declared to be ABSOLUTE at 174. The value cannot be changed once execution begins, since references to this variable will refer to word 174 in your field length rather than word 174 in your minus words. This latter method relies on a bug rather than a feature of the loader. Consult your local loader expert before you use this feature to be sure it has not been fixed.

BATCH, ORDER, and Your Controllee

For debugging purposes, your controllee should be designed to run alone or under ORDER; for production purposes, it should be designed to run under ORDER, which in turn is run under BATCH. There are two things to remember. One is that your controllee should be designed so it will not require messages to be typed in by a real person during production. The other is to understand how messages go back and forth between BATCH, ORDER, and your controllee.

BATCH sends four kinds of messages to ORDER. One is the execute message to start ORDER. A second is anything the operator types in (deliberately or otherwise) that BATCH does not recognize as a message for BATCH. Third are ORDER execution controls as described in the ORDER manual. Fourth is the message END. In any situation where BATCH is awakened (for example by an unexpected message coming up to it), it restarts ORDER's dropfile and sends it the message END. Notice that this END message is not preceded by a LINEFEED, so ORDER does not recognize it as a command to stop. All ORDER execution control commands start with a LINEFEED (except SW1., which may or may not have a leading LINEFEED).

ORDER sends four kinds of messages to your controllee. One is the execute message given on the *XEQMES card. A second is a restart message given on the *RSTMES card unless superseded by a call to SRSTMES. A third is the sense-switch message equivalent to flipping sense switch 1. (You should have the bit in GOBCOM flipped rather than rely on a message.) Fourth is anything that comes from BATCH (or the terminal if you are running ORDER yourself) that ORDER does not recognize as an ORDER execution control command. As mentioned before, this category includes stray messages typed in by the operator or the END message when BATCH does not know what else to do.

Between a controller and its controllee are two message buffers maintained by the operating system. For messages going up, there are various options for stacking, replacing, or ignoring messages, but for messages going down to the controllee, the current message replaces any message that the controllee has failed to pick up. An example of the problem you can create is to send a message to controller asking for input and then asking for a message from controller. When your controllee is run alone or just under ORDER, the input can be given on the execute line or on a *XEQMES card. The message asking for input goes out one system buffer, and the input given on the execute line or the *XEQMES is already sitting in the system input message buffer. When your controllee asks for this message, it picks up this message and is satisfied. However, when you run under BATCH, BATCH is awakened by the unexpected message. BATCH puts the message in its record file and restarts ORDER with an END message. Since ORDER does not recognize END as an ORDER command, it restarts your controllee with the message it received from BATCH. The END replaces the execute message in the input buffer, and your controllee does not get the message it was expecting. The solution is simple. Your controllee should try to get a message from controller first. If there is none, then send a message to controller asking for data. After that, do another get message from controller. Remember that ORDER normally passes to your controllee any messages it gets from its controller that do not start with a LINEFEED. The only exception is the SW1. message. The disposition of this message depends on what you specified on your *TIME card. ORDER is assuming that the operator meant to type LINEFEED SW1., but forgot the LINEFEED.

The case of messages going upward is simpler. ORDER asks the system to leave its output bypass open when it runs your controllee. The subroutines in ORDERLIB for ORDER job control always ask the system to close the bypass first so that their messages will be received by ORDER rather than be passed upward to the next level (i.e., BATCH). ORDER otherwise is awakened only by messages coming down from BATCH or error or completion messages notifying ORDER that your controllee could not continue or terminated normally. BATCH then copies any messages coming up into its RECORDFILE. Thus, messages ordinarily go not to the operator but into a dayfile for you to read later. Incidentally, BATCH's record contains only the first eight words of any messages that come up to BATCH. This record was not meant to contain large amounts of output.

We have not yet mentioned what to do if the system crashes while ORDER is running, because the user can do nothing about this problem. The hope is that ORDER itself is not going to be in memory long enough to be caught by a system crash. Therefore, most efforts are expended to see that ORDER's controllees can

be restarted. If BATCH sees that ORDER itself was the victim of a crash, it normally gives up on that job and starts ORDER off on another user's job. The reason is that user jobs can cause ORDER to crash even though nothing happened to the system. Since BATCH cannot be sure which one happened, it will not rerun the user's job. The only time BATCH will restart ORDER on your job is when ORDER died because of a disk-parity error. This error presumably cannot be induced by the user and would not occur again, so it is considered safe to retry your job. This means that your job had better be prepared to find that some working files or output files already exist and to realize that it is being run again, even though it may have successfully completed execution a few seconds ago.

MAINTAINING SOURCES AND BINARIES

Introduction

In maintaining a large production code you will have many subroutines containing many lines of code. Almost certainly you will have to modify routines or add new routines, and you will need something easier to handle than a giant text file. You will also want to keep your relocatable binaries, because compilation is expensive. When you change a subroutine, you will want to compile only the new routine and then replace the binary in your collection. You may also wish to save old versions of sources or binaries so you can back up to an earlier version if a new version seems to have bugs. Most of the serious production users that we know have developed maintenance systems to suit their own needs. In this document we will mention only the generally available utility routines on which you might build your own system.

Managing Sources

One way of maintaining sources is to put each subroutine in a separate file and use utility routine LIX to aggregate them into one large file. The subfiles can be easily added, replaced, or deleted by LIX. Transformations between a LIX format and a single giant text file may be accomplished with utility routine SFM, which in fact was designed for maintaining large source files by making each subprogram a separate file that can be stored in a LIX library file. The primary drawback of this system is that you are easily tempted into modifying and replacing a subroutine and not keeping a record of what you changed, should you need to undo the changes. If you use this system, you should seriously consider using alter files in the TRIX AC text-editing system to keep track of changes.

A more orderly approach is possible with utility routines MODIFY or UPDATE, which run under the SLOPE subsystem. MODIFY is smaller and faster than UPDATE but less versatile. Both of these utilities keep sources in a compressed form, squeezing out blanks. These utilities are line (or card-image) oriented, in that entire line images -- not just patterns within lines -- are added, replaced, or deleted. All lines, whether they be original lines or modification lines, receive an identifier and sequence number. Modifications are made in sets, and it is possible to activate or deactivate sets of modifications without deleting. Since line images are deactivated rather than deleted, it is possible to recover earlier versions by deactivating the recent modifications. MODIFY and UPDATE allow you to extract any or all decks for compilation and force you (for your own good) to introduce modifications in an organized manner. Another utility available only on the MFE network is the HISTORY system. It is a BCON

program for managing text files. Documentation is available through utility routine DOCUMENT.

Managing Binaries

LIBMAK is the tool for creating library formatted files of binaries. This format has a directory that assists the loader in finding the subroutines and thereby helps decrease the time required for loading. LIBMAK can also be used to add, replace, or delete routines from libraries. A binary library should be used only for current routines. If you keep superseded binaries, a LIX file is probably the best place to put them. Make each relocatable binary a separate file. Please note that the LIX file of binaries is not acceptable to the loader. In no case should binaries be kept in the form of punched card decks, since the MFE card reader does not read binary decks.

For overlay codes it is most convenient to group codes for a given code block into separate binary libraries, each library having precisely the codes needed for one of the code blocks. If you are running under ORDER, recompiling a subroutine, and loading it, be sure that the library containing the obsolete version is presented to the loader after the binary file containing the newly compiled version. You do this by putting the *LOD card after -- not before -- the *CHATR (*FORTRAN for Octopus users) card in the series of card images describing a code block to ORDER. If the *CHATR card does not point to an external file with the source, but actually precedes the card images of the source, the *LOD card should be placed after the last END card in the FORTRAN source. The *LOD card should then give the name of your library, followed (with no intervening blanks) by a comma. ORDER first compiles your FORTRAN with CHATR and assigns a name to the binary file produced by the compiler. ORDER then picks up the library name on the *LOD card and gives the two file names to LOD in that order. The comma must be appended to the library name, because the loader is looking for a purpose character at the right end of the character string to tell it what the rest of the string stands for. The comma indicates a binary file from which routines are used only if they are actually needed to resolve external references.

It is possible to put all your subroutines into one library even though not all of them are required in any one code block. The loader needs to know (among other things) the name of the entry point (a subroutine name) for each code block and what subroutine to force into the code block whose external references will in turn pull in the other subprograms needed in the code block. The name of the entry point must have a minus sign attached as a purpose character. ORDER will take care of this for you if you allow it to control the loading. The name of the subroutine to force in (almost always the same as the entry point name) is given without a purpose character appended. If this subroutine is present in source form to be compiled, ORDER will force it in. If it resides in your binary library, you should use the *LOD card to force it in. One problem arises in force-loading the main program out of a library. The main

program is always named MAIN., where the period is part of the name; however, the loader interprets the period as a purpose character. To force-load MAIN. out of a library, use the sequence of LOD commands,

b=bMAIN.bXYZb=bXYZb

where b means one or more blanks and XYZ is any symbol that does not conflict with the name of anything else. These commands equate XYZ to MAIN. and force-load XYZ. Since XYZ stands for MAIN., MAIN. will be force-loaded.

COMPASS

COMPASS subroutines are not susceptible to maintenance by SFM. You can still use LIX, MODIFY, or UPDATE, however.

COMPASS

Subroutine Linkages

To write in COMPASS, you will need to know the subroutine linkages used by CHATR (or CHAT). For five or fewer arguments in a subroutine call, the addresses are placed in registers X1 to X5, starting at X1. The number of arguments is placed in X6, and the return address is placed in X7. Jump to entry point plus one for subroutines and for integer and single-precision functions. If you are calling a double-precision or complex function, jump to entry point plus two words. Results of integer and single-precision functions are left in X6. Double-precision and complex functions leave answers in X6 and X7. If the subroutine call contains six or more arguments, all argument addresses should be placed in memory immediately before the return address, one address per word. At the entry of your subprogram you should start the first executable instruction the word after the entry location for subroutines and for integer and single-precision functions; you are not supposed to jump to and execute code right at the entry point. For double-precision and complex functions, allow two words and start your first instruction at the second word after the entry location. Currently there is a bug in the loader that requires the main entry point of a subprogram to be within seven words of the beginning of the subprogram. (Do not put a lot of constants at the start of the program.) Alternate entry points may be put anywhere. A main entry point is required and is characterized by being the same name as that specified on the IDENT card. CHATR does not use the return jump instruction. You must save the return address yourself. CHATR normally creates a variable named IENDIG. (the period is part of the name) to hold the return address when it compiles a subprogram. You should do likewise when you write in COMPASS. The COMPASS assembler can put out type-26 cards for use by the loader in constructing symbol tables; however, the location field symbols are all designated as integer variables. There are no argument, array, real, or other attributes possible, so when using DBCTRL or DEBUG, for example, you must be careful when examining values of arguments.

Instruction Timing

The timings of the various instructions are given in the 7600 hardware reference manual. The manual does not point out clearly, however, that there is such a thing as a trunk conflict, which must be avoided. There is only one path into the X registers, so only one of the eight X registers can receive a result in any given minor cycle. For example, a normalize cannot be issued immediately after a floating-point add, because the instructions require 3 and 4 minor cycles, respectively, and would return results on the same cycle to the X registers. The issue of the normalize will therefore be delayed. If you are

writing in COMPASS, you will need to know that there are four priority levels for returning results to X registers. Floating-point divide has the highest priority. Once the divide is issued, its result will come back in 20 minor cycles. SSM references have second priority. An SSM read instruction will issue as soon as it can, but without regard for memory bank conflicts. The storage-address stack will delay the delivery of the word from memory to the X register, if necessary, to avoid conflict with the result of a divide. Third in priority are the rest of the CPU instructions. Their issue is delayed so that they will not conflict with the result of a divide or the reading of a word from SSM. Last priority goes to an LCM read. This instruction is issued without regard to bank conflicts. The word read from LCM is delayed as long as necessary until there is a cycle where no other data is moving into an X register.

NOTICE

"This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Department of Energy, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately-owned rights."

NOTICE

Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Department of Energy to the exclusion of others that may be suitable.

Printed in the United States of America
Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161
Price: Printed Copy \$; Microfiche \$3.00

Page Range	Domestic Price	Page Range	Domestic Price
001-025	\$ 4.00	326-350	\$12.00
026-050	4.50	351-375	12.50
051-075	5.25	376-400	13.00
076-100	6.00	401-425	13.25
101-125	6.50	426-450	14.00
126-150	7.25	451-475	14.50
151-175	8.00	476-500	15.00
176-200	9.00	501-525	15.25
201-225	9.25	526-550	15.50
226-250	9.50	551-575	16.25
251-275	10.75	576-600	16.50
276-300	11.00	601-up	— ¹
301-325	11.75		

¹Add \$2.50 for each additional 100 page increment from 601 pages up.

Technical Information Department

LAWRENCE LIVERMORE LABORATORY

University of California | Livermore, California | 94550