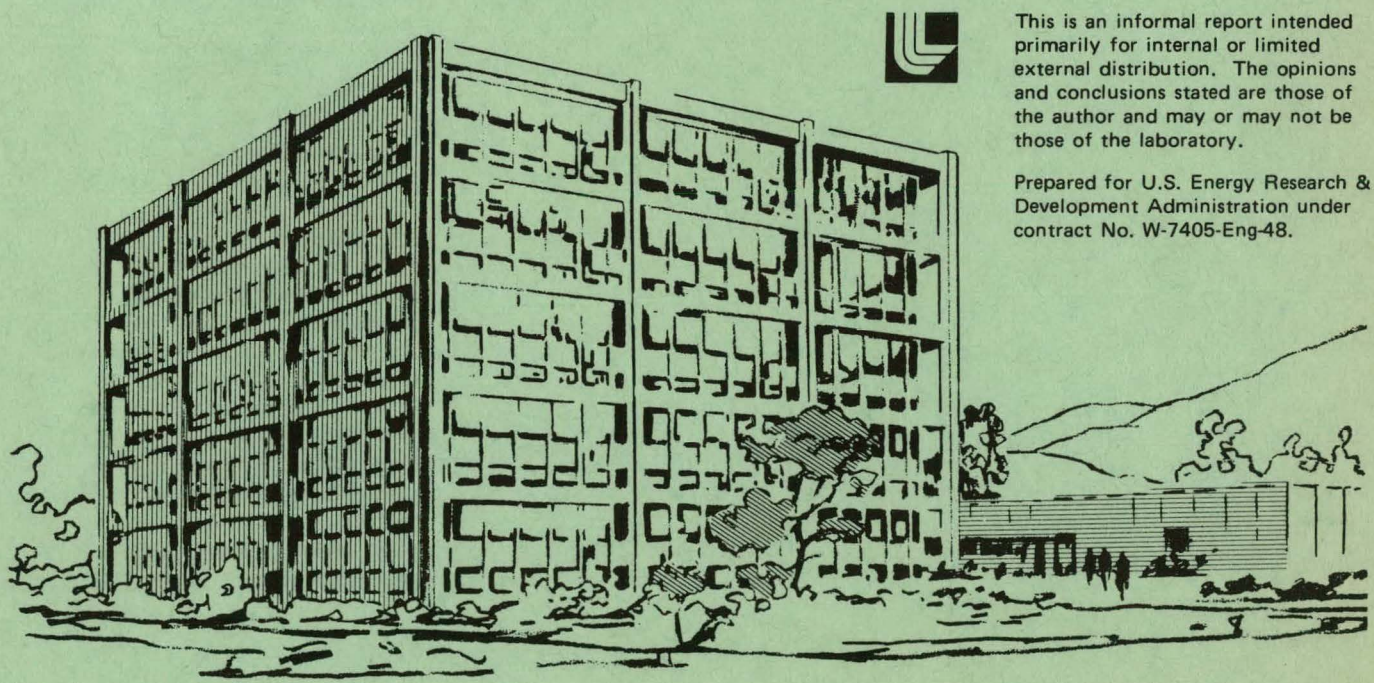
UCID- 17556

Lawrence Livermore Laboratory

AN LTSS COMPENDIUM: AN INTRODUCTION TO THE CDC 7600 AND THE LIVERMORE TIMESHARING SYSTEM

K. W. Fong

August 15, 1977



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

DISCLAIMER

THIS PAGE

WAS INTENTIONALLY

LEFT BLANK

AVAILABILITY

This document is available online as follows:

```
ELF RDS .717675:UCID:UCID17556 / 1 1
```

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

```
TRIX AC / 1 1
.PRINT(<NIP UCID17556 BOX ann identification>)
.END
```

CONTENTS

AN LTSS COMPENDIUM: AN INTRODUCTION TO THE CDC 7600 AND THE LIVERMORE TIMESHARING SYSTEM

ABSTRACT

This report is an introduction to the CDC 7600 computer and to the Livermore Timesharing System (LTSS) used by the National Magnetic Fusion Energy Computer Center (NMFECC) and the Lawrence Livermore Laboratory Computer Center (LLLCC or Octopus network) on their 7600's. This report is based on a document originally written specifically about the system as it is implemented at NMFECC but has been broadened to point out differences in implementation at LLLCC. It also contains information about LLLCC not relevant to NMFECC. This report is written for computational physicists who want to prepare large production codes to run under LTSS on the 7600's. The generalized discussion of the operating system focuses on creating and executing controllees. This document and its companion, UCID-17557, *CDC 7600 LTSS Programming Stratagems*, provide a basis for understanding more specialized documents about individual parts of the system.

INTRODUCTION

This document is an adaptation of the first of two reports that I wrote for the National Magnetic Fusion Energy Computer Center (NMFECC) on-line documentation system on the subject of applied programming. The original documents were aimed specifically at users of the MFE 7600, which runs a slightly different set of software than do the CDC 7600 computers that are a part of Lawrence Livermore Laboratory's Octopus computer system. Therefore, most but not all of the material is applicable to Octopus as well. This UCID and UCID-17557 bear the same titles as the NMFECC documents from which they are derived. Almost all the MFE material has been retained. The difference is that I have added information about 817 disks, the Octopus charging algorithm, and the Octopus hardware configurations, so that Octopus readers may have a complete picture of their own system. The NMFECC has received considerable help from Octopus personnel and hopes to return the favor in part by sharing this educational material it has prepared.

This document and its companion, UCID-17557, were written for computational physicists who want to prepare large production codes. I contend that a knowledge of FORTRAN alone is not sufficient for achieving this goal. You must also learn some applied programming. These documents attempt to give an overall view of the 7600 and LTSS so that you may profitably read the many other specialized documents about parts of the system. This report deals with the machine and its operating system. UCID-17557 discusses the problems to be addressed in applied programming and some techniques for dealing with them.

LTSS is an exceedingly flexible system. Since there is usually more than one way to solve a problem, your goal should be to understand the system and its software well enough that you can easily find the cheapest acceptable solution for your problem. This understanding comes with writing, compiling, loading, executing, and debugging moderately complex programs. It cannot come solely from reading these documents; nevertheless, I feel that these documents contain much useful information distilled from the experience of many other users as well as myself and that reading these documents should accelerate your learning. I would even say you should read these documents before proceeding to work on a very large production code.

THE HARDWARE

Memory

The NMFECC's 7600 has two memories. One is called small semiconductor memory (SSM), and the other is large core memory (LCM). Memory sizes are measured in words rather than bytes because the 7600 is a word-oriented machine. A word is 60 bits long. SSM contains 65,536 (decimal) words; however, not all of this is available to you. The operating system needs part of it and allows you to have at most 157,760 (octal) or 57,328 (decimal) words. SSM is what you normally think of as the computer's memory. It can hold both data and instructions and is the memory most easily accessible to the central processing unit (CPU). LCM is a secondary memory that is used for holding data. LCM is also used for input/output (I/O) operations in that all information going to and from disk must go through LCM. Instructions exist for transferring data between SSM and LCM. The size of LCM is 512,000 (decimal) words; however, part of it is reserved by the operating system, so you may have at most 1,414,600 (octal) or 399,744 (decimal) words. We will see later that there are additional constraints on the amount of LCM that you can or should use. The Octopus 7600's are similar to the NMFECC 7600 except they have small core memory (SCM) instead of SSM. In the rest of this document, references to SSM are also applicable to SCM except where stated otherwise.

Central Processing Unit

Computation occurs in the central processing unit (CPU). Viewed simply, the (CPU) has three parts of concern to the user: (1) registers, (2) functional units, and (3) instruction stack.

The 7600 is a register-oriented machine. All data must be fetched from memory to a storage location (i.e., a register) inside the CPU before they can be used. The registers are also used to calculate the addresses from which operands must come in memory and addresses to which results should be stored in memory.

The functional units accept operands from registers and return results to registers. Most of the functional units are segmented so that another pair of operands may be fed in even before a preceding pair has produced a result. The unit thus acts like an assembly line. In addition, different functions are handled by different functional units; thus, a floating-point multiply may be initiated immediately after a floating-point addition is started, without waiting for the addition to complete. Therefore, great speed of computation is possible if instructions are carefully sequenced and registers are cleverly

allocated to take advantage of the potential concurrency. We shall see later how you may do this.

The 7600 executes code by reading instructions from memory into its instruction stack and then executing the instructions. Instructions may be either 15 or 30 bits long, so it is possible to pack two-to-four instructions per word. A 30-bit instruction is not allowed to straddle a word boundary, so it may be necessary to insert no-ops (non-operations) into the instruction stream for padding. The instruction-word stack consists of 12 words, two of which are look-ahead words. In parallel with the instruction-word stack is the instruction-address stack (also 12 words long), which contains the memory address from which the corresponding word in the instruction-word stack came. The look-ahead words are pre-fetched in the hope that you will execute instructions sequentially in memory. This minimizes delays waiting for another instruction word to be read from memory. The instruction stack also keeps the nine words that were read previous to reading the current instruction word, so a jump to a very recently executed instruction is a jump to an instruction already in the stack. Thus, if a DO loop is short, it can be represented by a series of instructions contained entirely in the instruction stack. This is a desirable situation because instructions going to the stack and data going to the registers must both come from memory, and a memory-bank conflict could result if an instruction and a datum are both in the same memory bank. To appreciate this, you must know that SSM is laid out in 16 banks in which consecutive words reside in consecutive banks; however, the 17th word wraps around and is in the same bank as the first word. Thus, when a bank is asked to deliver a second word before it has recovered from delivering the first, a bank conflict occurs, and the second word is delayed. SCM, unlike SSM, is laid out with 32 banks, so bank conflicts occur every 32 words instead of every 16. Again, we will see later what can be done to avoid such delays.

Floating-Point Numbers

In an oversimplified view, the left 12 bits of a floating-point number represent the exponent to the base 2, and the right 48 bits are the coefficient. Since the FORTRAN compiler takes care of representing and normalizing floating-point numbers, users of FORTRAN generally need not be concerned with these topics. What is important is that a single-precision number has approximately 14 decimal digits of precision, so that double precision (which is very expensive on the 7600) is generally not needed. The range of floating-point numbers is approximately 1.0E-293 to 1.0E+322.

There are two non-standard floating-point operands: overflow and indefinite. Normally, using or generating either of these in a floating-point operation aborts your code. (Dividing zero by zero is the most frequent error that generates an indefinite.) Normally, an underflow condition returns a zero result but does not interrupt your code; however, it is possible to run your

code in a mode where an underflow does cause an interrupt. Consult the 7600 hardware manual for more detailed information.

Exchange Jumps

Every user program has a set of 16 words, called an exchange package, associated with it. Whenever the user's program must stop, the contents of the CPU registers are written into the exchange package. (For the time being, we will not discuss the physical location of the exchange package.) The contents of the exchange package are validated by the operating system and then copied into the registers if and when the program regains the use of the CPU. One reason that a program might stop is that it created an overflow or indefinite. In this case, the hardware stops it automatically. Another reason is that it has voluntarily given up the use of the CPU for some reason. In particular, no user program is allowed to deal with files or perform I/O directly. Instead, the program leaves a request for file manipulation or I/O in one of the locations where the system looks for such requests and then performs an exchange jump. The exchange jump causes the program to relinquish the CPU to the operating system and also causes the registers to be saved in the exchange package. The operating system services any legitimate requests and then allows the user program to resume when appropriate. One of the items in the exchange package is the program counter (P counter), which is the address of the instruction word to be executed when the program resumes. When a program is aborted because of some error condition, the exchange package is saved, and you may use various utilities to determine the final value of the P counter. This value is the location where the program stopped unless the error was an erroneous jump.

The process of doing exchange jumps in order to make requests of the operating system is known as making system calls. System calls are seldom coded directly by the user. Usually you use subroutines in an existing library to take care of your I/O and other system requests. In a large production code, however, an intermediate course is sometimes used. In this situation, the programmer has designed his own I/O scheme in terms of the basic functions provided by the operating system. He then uses subroutines (such as GOB, FROST, SETIO, and REIO in ORDERLIB) to construct the system call and do the exchange jump. We have found that users who obtain good I/O performance on LTSS have had to descend to a level just above stringing together the bit fields in a system call. BLIB76 and URLIB are other libraries in which you might find subroutines for issuing system calls.

You may feel that groveling in such details is inappropriate for a computational physicist. If a code is to be used only once, it is economical to use the inefficient standard FORTRAN I/O routines, but if a production code is to be run repeatedly and uses dozens of hours of 7600 time, then it is worth designing and programming it to take every advantage of the system. You cannot expect a common FORTRAN program to run efficiently on all machines and systems.

Large codes must be tailored to the machine and operating system on which they will be run. If correct design means the difference between running for one hour or ten hours, then it may also mean the difference between running or not running at all.

Input/Output Capability

I/O is the area most often ignored by the unsophisticated programmer to the detriment of his pocketbook. We will consider the influence of the operating system later, but here let us look at what the hardware can do. Data from disk can go only to LCM. If the data are needed in SSM, they must then be copied from LCM to SSM. Likewise data can be written to disk only from LCM. If the data are in SSM, they must first be transferred to LCM. We will see that an excessive amount of inadvertant copying between SSM and LCM is easy to do but also easy to avoid. Once a data transfer has started between LCM and disk, it is possible to return to your program and compute while data are being moved. Obviously, you should not use data while they are in transit. You should read in data before you need them and meanwhile compute on something else. Just before you need the data, test to see if the data transfer is complete and give up the CPU only if the transfer is not complete. In fact, it is quite common in a production code for two I/O operations to proceed simultaneously with central processing. This minimizes the wall-clock time needed to run the job. Since the 7600 can crash like any other computer, it behooves you to push your job through as quickly as possible. Furthermore, the present charging algorithm contains a charge proportional to the length of time you spend in the 7600, so you are penalized if you fail to overlap I/O with CPU when you could have.

Making a few large I/O transfers is usually cheaper than making many small ones. The usual mechanism for consolidating small amounts of data is an LCM buffer. If the LCM space is available, pack small logical records into one large physical record before writing. Positioning the disk for a read or write is expensive, so you should move large blocks of data to keep the cost per word down. Correct buffering can make a tremendous difference in I/O charges and execution time.

Controllees

All executable programs must start as an executable core image on disk. Executable programs are usually called controllees and are constructed on disk by the loader. To execute a controllee you normally type the controllee's name at your terminal. (In this case you are the controller of the controllee.) Next you type execute-line parameters if any, a slash, the running time, the value, and a carriage return. The program is not through until the "all done" message comes back to your terminal. Before the program completes, it can send messages to your terminal and receive responses from you. All these interactions with the outside are done via system calls. If the program wants to send you a message, it issues a system call, "send message to controller." The system picks up the message and prints it at your terminal if you are logged in under the suffix under which the program is running. The controller is whatever started the controllee. In this case, it is the terminal, which for historical reasons is referred to as TTY. The designation as controller or controllee is relative, however. If you execute program A from your terminal, you are the controller, and A is the controllee. But on LTSS, A is allowed to execute a program, so if A starts up program B, A is the controller of B, and B is the controllee of A. Controller-controllee chains may extend to ten levels.

Let us consider an example of where the controller-controllee concept may be used. The initialization and clean-up phases of many production codes require extensive programming. Each of these is coded as a separate controllee, as is a third controllee that actually does the calculations. Then, to execute the three controllees, you write a controller or use an existing general-purpose controller such as ORDER or BCON). The first controllee will create and initialize disk files for the second to use. The third will save files, construct plots, and dispose of files. This scheme has at least two virtues. One is that, after the files and contents are clearly defined, separate people can write the separate controllees without needing extensive co-ordination. A second is that overlaying of codes can be avoided. It may also be easier to debug three smaller codes than one big one.

How is a controllee laid out on the disk? The first thing in the disk file is 230 (octal) words called the minus words. The system keeps a copy of these minus words in its share of LCM and records in it information about your job and the files that are connected to it. After the minus words comes word zero of your program. Starting at your word zero is the image in SSM of your program. Your SSM image contains code, labeled and numbered common blocks, and the space needed by your largest combination of overlays. The size of this image is called field length small (FLS). This is followed by your LCM common blocks out to field length large (FLL). The LCM image always contains the SSM image, so FLL is always greater than or equal to FLS.

To execute your job, the system reads your LCM image into LCM and then copies that part of it which represents the SSM image into SSM. The minus words contain the exchange package, which is then copied into the registers. Since the loader has set the P counter at the beginning of your program, that is where execution begins. The loader also sets the mode flags for overflow and indefinite in your exchange package. This lets the system know that you want to be stopped as soon as you generate an overflow or an indefinite. The mode flag for underflow is not set by the loader; therefore, the system lets you continue running without an interrupt whenever you generate an underflow. This is usually harmless since an underflow is returned as a zero, which can be used in further calcuations. Overflows and indefinites have a special representation and cannot be used as operands in subsequent floating-point computations.

In the course of running your program, the system may need to copy your job to disk. For example, some other user could bid a higher priority. His job will run while you are temporarily restrained from executing. If the system needs the memory space, your memory image will be dropped into a disk "dropfile." If you have not issued a system call to tell the system otherwise, your dropfile is the original controllee disk file. Dropping to it destroys the original copy of the program so that it cannot be executed from the beginning again. Therefore, you will usually call CHANGE or ADJUST (in ORDERLIB), which make system calls to create a separate dropfile. This call should be the first executable instruction in your program so that the system finds out you want a separate dropfile before -- not after -- it has to drop you to disk the first time. If you allow the ORDER batch-processing system to load your program, it will load in extra code to perform the system call for you to drop to another disk file (i.e., you don't need to call CHANGE or ADJUST).

Note that the dropfile is actually a copy of your program at some intermediate state of execution. In case of a system crash where the system forgets what was supposed to be running, you can give the dropfile name on the execute line to restart the program. This usually works if the program was only reading or only writing each of its disk files, but it may fail if any file is being both read and written. The failure can occur in the following way. Suppose a job drops to disk and later continues execution. When it is back in memory and executing, it reads a disk file and then overwrites it with new information. Then the system crashes and loses the job in memory. At this point, the dropfile that is left represents a state before the file was modified. If it is restarted, it will read new rather than old information. If it does not recognize that the wrong information was read, it may generate erroneous results. A good production code allows for system crashes by telling ORDER what controllee to start after a crash in order to verify the existence and integrity of all working files before resuming computation. Restarting the dropfile of a program that was using tapes is usually not possible, because tapes are usually dismounted during a system crash. It is possible to write a recovery procedure for such codes, but you are advised instead not to connect tapes directly to your program.

Timesharing and FLL

LTSS is a timesharing system. This means each job gets time slices that are usually less than it needs to run to completion. There are two kinds of time slices: one for LCM residence, which depends on FLL and priority, and one for SSM, which depends on priority only. If the job is not finished at the conclusion of a time slice, it will get another slice later (unless the system has no other job to run, in which case it can have another slice immediately). When your job has exhausted its current slice, some other job will be given a chance to run for a while. What happens to your job depends on the other jobs. If the FLS for your job plus the FLS for the other job or jobs fit into SSM, your SSM image may be left in SSM while other jobs compute. If other jobs need more SSM, the system will copy your SSM image into that part of your LCM field length reserved for your SSM image. If the system needs your LCM space for someone else, your LCM image, including the SSM image, will be dropped to disk (into your dropfile). When you will get back into memory to resume depends on what priorities other users have bid that are higher than yours and how large other jobs are that are bidding the same priority as yours. For any given priority, the system selects the smallest job. That is why jobs with very large FLL cannot get in very often without a high priority when the system is heavily loaded. You should be very careful about writing a program that uses all available LCM because you will have a hard time running it during the day to debug it. Furthermore, if such a job does unoverlapped I/O, the CPU will become idle because no other job can come into LCM simultaneously to use the CPU. This is why the charging algorithm is designed to penalize jobs that do not overlap their I/O and CPU.

Using Tapes

The use of tapes by user jobs running in a timesharing mode is discouraged because a job might tic up a tape drive indefinitely. Try to use FILEM on the MFE network or ELF on the Octopus network instead of tapes. If you absolutely must use tapes, use utility routines such as ADT, RDFILES, and MCT to write and read tapes. These routines will copy from disk to tape or tape to disk and relinquish the tape drive as soon as possible. Utility routine TAPECOPY may also be used for tapes coming from or going to other computer systems.

Using Disks

You should think of disks as the only I/O devices your job should use. Print and film files should be thought of as disk files that will later be processed through the appropriate device. On LTSS disk files are of fixed size. The size is specified at the time of creation. Many existing routines will create families of files. When one member of a family is filled, another file with a related name is created, and output continues there. A file may be truncated if the output does not fill it, but a file cannot be expanded. Disk files are allocated on contiguous disk sectors. If you ask for too large a file, you may not find enough unflawed consecutive sectors to accommodate it.

Disk files fall into two general categories: public and private. The files you normally create are private files under your own user number. Unless you save them on FILEM (or ELF), they will be purged after some period of disuse. This is to prevent the 7600's disks from being clogged with unused files. In order to allow production computing more disk space, you should destroy unneeded files before you go home for the evening. Public files are not purged. They are used to hold libraries, utilities, and other useful things where they will be readily accessible to users. For example, the CHATR (or CHAT) compiler and the LOD loader are public files that anyone may execute. Compiling and loading are considered user functions, not system functions. You will notice, for example, that CHATR drops to +QUIKTRAN in your private file space.

The NMFECC's 7600 has two kinds of disks available: CDC models 819 and 844 disks. Four of each are accessible by users. Both kinds of disks rotate at approximately 17 milliseconds (ms) per revolution. The maximum time required to move the heads to the correct position is 85 ms for 819 disks and 55 ms for 844 disks. The bandwidth or transfer rate of the 819 disks is about 30 million bits per second, so 1000 (octal) words can be moved in about 1 ms. The 844 disks are slower and require about 6 ms to transfer 1000 (octal) words. For small amounts of data, the 844 disk is a better place because its shorter seek time (positioning the heads) is far more important than its transfer rate. Files from which many thousands of words will be accessed at a time should be put on the 819 disks because the transfer time will dominate the seek time. Octopus 7600's also have CDC model 817 disks, which we will describe later.

On the MFE 7600, data move between LCM and disk files under the control of peripheral processing units (PPU's), which are computers in their own right. The disks are given unit numbers. The PPU's also are numbered. The 819 disks are units 1, 2, 3, and 4 and are served by PPU's 4, 5, 6, and 7. Because the transfer rate of the 819 disks is so high, two PPU's are required to effect a read or write. One 819 can be accessed by only one pair of PPU's at a time. This means that two 819 files that are to undergo reading or writing simultaneously must be on separate units. Also, no more than two 819 files may be read or written simultaneously, since there are only four PPU's connected to the 819 disks. PPU's 4 and 5 can access any of the 819 disks, as can pair 6 and 7. Your program, of course, may be connected to as many as sixteen 819 files at

a time, but your read and write requests will have to be queued up once all four PPU's are busy. The 844 disks are units 5, 6, 7, and 10 (octal). They are serviced through PPU's 2 and 3, either of which can access any of the 844 disks. Only one PPU is needed to read or write an 844 disk, since the transfer rate is low enough to be handled by a single PPU. Again, two files on the same unit cannot be read or written simultaneously; the requests must be queued. The two PPU's can service two requests at any moment; other requests to read or write 844 disks must be queued. At the system-call level, you may specify which unit (number) you desire when you create a file. If the system cannot find enough space on the unit you requested, it will next look at a unit of the same type--i.e., another 844 or another 819. If that fails, your file is created wherever space is available. At the completion of a file-create call, the system will tell you which unit actually was used. This detailed information is hidden from you when you allow ORDERLIB subroutines to make the create calls for you; however, it may be important to a production code to place its working files on disk of a specific type.

In estimating the elapsed wall-clock time required for an I/O operation, you should add 150 ms to the transfer time. The 150 ms not only cover the seek time and rotational latency, but allow a generous delay while the operating system completes other users' I/O requests that arrived before yours. That is, if some other user is reading from the same disk unit that you want to read, your request is held by the system until the unit and an appropriate PPU are available to process the request. In the meantime, you should try to find useful computation to perform for 150 ms plus the transfer time. That computation should neither use the data being read nor reuse the space from which data are being written until the I/O operation is complete. The actual wall-clock time required to complete I/O clearly depends on the system load. The recommended base of 150 ms was derived from looking at the MFE 7600 statistics on a busy day.

The sector size of the 819 disk is 512 (decimal) or 1000 (octal) words. There are 20 (decimal) sectors per track and 200 (decimal) sectors per cylinder. The 844 has 64 (decimal) word sectors with 24 (decimal) sectors per track and 456 (decimal) sectors per cylinder. Disk file addresses always start at zero, and address zero is always on a sector boundary. The basic system call for reading or writing disks allows you to transfer an arbitrary amount of data to or from any starting address on disk; however, certain disk starting addresses and transfer sizes are preferred above others. If you begin writing a block of data in the middle of a sector and end in the middle of another sector, you have the worst possible case. The sector where you start writing must be read into the PPU buffer, and your data is written over it. Then the revised copy of the sector is written back out to disk. Intermediate sectors go from LCM through the PPU buffer to disk. The last sector must be read from disk to the PPU buffer, where part of it is overwritten with the remaining data from LCM, and then it is written back to disk. What we have described is the preservation of the unreferenced portion of the first and last sectors. Notice that each causes you to lose a disk revolution.

There are two solutions. One is to write multiples of sector sizes, starting on sector boundaries so that there are no unreferenced portions. The

other is to use special options in the system call to tell the PPU not to preserve the unreferenced portions of the first or last sectors. In the latter case, the unreferenced portion of the sector is overwritten with whatever happens to be in the PPU buffer. On the Octopus 7600's, which may have classified data, the PPU must generate disk pattern in that part of its buffer that you don't use, so that you will not pick up a chunk of someone else's possibly classified data. The PPU generates this disk pattern so slowly that you are almost certain to lose a disk revolution; therefore, the option not to preserve the unreferenced portion of a sector is generally not economical. You should instead try to write multiples of sector sizes, starting on sector boundaries. When reading a disk file, there is no problem with preserving sectors, but you should be aware that entire sectors must be read whether all the data in them are needed or not. Therefore, it is marginally more economical to read multiples of sector sizes, starting on sector boundaries.

The Octopus 7600's use 817 disks as well as 819 and 844 disks. The 817 disks come as two units in a single cabinet. Each cabinet contains two spindles with 32 recording surfaces per spindle. You might expect that each spindle is a logical unit, but this is not so. The top 16 recording surfaces of the spindles are one unit, and the bottom 16 recording surfaces of the spindles constitute the other unit. Each unit has its own set of heads. Since a unit is split between two spindles, a set of heads must likewise be split into two stacks, one for each spindle. The two stacks of heads are mechanically linked so that, when one stack is in a given cylinder for its spindle, the other stack must be in the corresponding cylinder of the other spindle. The 817 disks have 40 sectors per track, and each sector holds 512 words of data. There are also 512 cylinders in a spindle. All together, an 817 cabinet offers 83,886,080 words of storage. Data are read and written with eight bits in parallel. As you write to disk, eight bits go to the upward-facing recording surfaces on eight consecutive platters on a spindle. At the completion of a revolution, you have written 20,480 words. You then write eight bits in parallel on the downward-facing recording surfaces of eight consecutive platters in the same cylinder on the same half spindle. Thus you can write up to 40,960 words in any particular cylinder on any half spindle. If you have more data, you go over to the corresponding cylinder on the other spindle to write 20,480 words on the upward-facing surfaces and then 20,480 words on the downward-facing surfaces. Thus, you can write up to 81,920 words before you have to move the heads. Since there are 512 possible head positions (or cylinders), a logical unit will hold up to 512 x 81,920 = 41,943,040 words. It takes 30 ms to move the heads to an adjacent track. The maximum positioning time for the heads is 145 ms. The average random-positioning time is 85 ms. The 817 makes one revolution in 34 ms. Forty megabits per second is the highest instantaneous-data-transmission rate, and 36 megabits per second is the average rate.

The admonition to move data blocks that are multiples of the sector size and that start on sector boundaries also applies to the 817 disks. Like the 819, the 817 has 512 word sectors, but, unlike the 819, the 817 can write only 160 sectors for a given position of the heads. As of August 1977, the R, S, U, and Z machines each have eight 844 disks, which are logical units 5 to 14 octal, and all the 844 disks are accessible only through PPU 2. Thus, the Octopus 7600's cannot access two 844 files simultaneously. The 817 disks are used with

machines R, S, and U. Logical units 1 and 2 in one 817 cabinet are served by PPU's 4 and 5, and logical units 3 and 4 (also 817's) are served by PPU's 6 and 7. PPU's 4 and 5 cannot access units 3 and 4, and PPU's 6 and 7 cannot access units 1 and 2. As was the case with 819 disks, the 817 transfer rate is so high that a pair of PPU's is needed to handle a transmission. If you expect to do simultaneous transfers to two 817 files, you must place one file on units 1 or 2 and the other on units 3 or 4. Machines R, S, and U do not have 819 disks. Machine Z does not have 817 disks; instead, it has four 819 disks configured exactly the same way as the NMFECC 7600.

Connecting Disk Files

In order for a program to use a file, it must be connected to the file. Each program has up to 16 I/O connectors (IOC's) that it can use. They are numbered 0 to 15 (decimal). Each IOC corresponds to a three-word block in the minus words. The system uses these to record which files you are connected to. You can become connected to a file by creating the file or by opening an existing file. There are also system calls for closing or destroying files.

When you do BCD I/O (formatted reads and writes in FORTRAN) and use the PROGRAM card, the opening of input files and creation of output files is handled automatically by the BCD read/write routines in ORDERLIB. For FORTRAN binary (unformatted) reads and writes, existing files to be read will be opened automatically, but files to be written must be created explicitly. For any other I/O, you must handle the file openings and creations yourself. Routine DEVICE in ORDERLIB may be used to create files, and routine ASSIGN may be used to associate logical unit numbers with file names. The PROGRAM card causes buffers to be reserved in a common block in LCM named IOCHIP$. Since these buffers are used only by the BCD and binary read/write routines, you need not mention on the PROGRAM card which files will be accessed by BUFFER IN/OUT, or other sets of I/O routines.

File Structures

The various file structures used by ORDERLIB are described in a short section at the beginning of the chapter on I/O routines in the ORDERLIB manual. This section is only three pages long, and you should read it to understand the terminology we will be using. The BCD read routines can read either packed ASCII or squoze-monitor files. The BCD write routines write only packed ASCII files. The binary tape-simulated files are an archaic format read and written

only by the binary read/write routines. This format is a relatively expensive one to read and write binary data, and you should avoid it. This means avoiding FORTRAN binary read/write statements. The so called two-argument BUFFER IN and BUFFER OUT statements read and write squoze-monitor files. Bits are transmitted between memory and disk without character conversion. The two-argument BUFFER routines are comparable in cost to the binary read/write routines. Better are the three-argument BUFFER routines where you specify the disk address yourself. The three-argument mode does not depend on any file structure. It essentially allows random access so that you can read or write exactly what you want. You should not settle for anything less than the three-argument BUFFER IN/OUT for your scratch files in a production code. The use of BUFFER IN and BUFFER OUT is described in the CHATR manual and in the CHAT manual (LTSS-207). In a sense, the BUFFER routines are misnamed, since there is in fact no buffer. The data to be transferred should reside in LCM. If the data happen to be in SSM, the system will block copy your entire SSM image to LCM before doing the transmission and then block copy you back to SSM. This means you cannot continue computing with the CPU while the data are moving. You should block copy the data between SSM and LCM yourself to minimize your cost. Methods for block copying between SSM and LCM are described in the CHATR and CHAT manuals.

System Documentation

ORDERLIB is considered to be the system library; however, you are not required to use it. Unlike other operating systems, LTSS permits you to use your own I/O routines if you desire. Indeed, users often have legitimate reasons for not using ORDERLIB. We cannot decide, without knowing your problem, what alternative you should use to supplement ORDERLIB. You should read the chapter of ORDERLIB describing the I/O routines and the appendix demonstrating the use of these routines to learn the capabilities and limitations of the ORDERLIB routines. ORDERLIB also has some routines, such as GOB, FROST, SETIO, and REIO, to help you write your own I/O routines. Another public library is URLIB, where subroutines are available for making system calls. A further source of useful I/O routines is public library BLIB76. Finally, the SYSCALLS document (or LTSS Chapter 10) describes all the elementary services the system can perform for you and how to request them.

Utility Routines

LTSS is not a control-card-oriented system with extensive job-control language. Services such as compiling, assembling, loading, reading, and writing tapes are performed by controllees that happen to be in public files so everyone can use them. Even ORDER is a fairly normal controllee. Although ORDER looks

to you like a batch processor, it looks to the operating system like just another user program. Text editors are also utility routines.

Charging Algorithms

To estimate the cost of various programming strategies, you will need to know the algorithm used to calculate your bank-account charge. We start first with the NMFECC charging algorithm.

```
CHARGE TIME = (UNWEIGHTED CHARGE TIME)*(PRIORITY)

UNWEIGHTED CHARGE TIME =
     (CPU FACTOR)*(CPU CHARGE)
    +(I/O FACTOR)*(I/O CHARGE)
    +(MEMORY FACTOR)*(MEMORY CHARGE)

I/O CHARGE = (I/O CHANNEL TIME) +
     (VOLUNTARY LOAD/DUMP TIME)

MEMORY CHARGE = (CPU CHARGE)
    +(I/O IDLE TIME)
    +(VOLUNTARY LOAD/DUMP TIME)
```

CHARGE TIME is the amount of time deducted from your bank account.

PRIORITY is the quotient of the value and time as specified on the execute line. The default is 1.0.

CPU CHARGE is the real CPU time used by your program, including CPU time needed by the operating system to service your system calls. Most system calls are charged the actual CPU time required, but some are charged at a flat rate.

CPU FACTOR is currently 0.8.

I/O CHANNEL TIME is the time a PPU (or pair of PPU's for an 819 disk) requires to process an I/O request. It does not include queue time--the time between the issue of your request to the system and the time the system issues your request to the PPU. I/O channel time is measured from the instant the system sends the request to the PPU until the instant the system receives the final status message from the PPU. By allocating disk files appropriately, it is possible to use up to four I/O channels simultaneously. A tape operation could cause a fifth channel to be used simultaneously. The time for each channel is added up in the I/O CHANNEL TIME, so that, if several channels are used simultaneously, the I/O CHANNEL TIME will exceed the actual elapsed wall-clock time.

VOLUNTARY LOAD/DUMP TIME is incurred whenever your program does something that requires it to be dumped to disk or be read in after a voluntary dump. Sending and getting messages to and from controllers and controllees are examples of a program voluntarily going to disk. A load charge is always incurred when a program is initially loaded into memory for execution. Upon completion of execution, a dump charge is incurred only if the dropfile must be saved. Normally the dropfile is destroyed and the core image is erased upon termination. Both the load and the dump times are calculated at the rate of 17,000 microseconds plus 2 microseconds for each word transferred. The number of words transferred is FLL plus 230 (octal).

I/O FACTOR is currently 0.2.

I/O IDLE TIME is the amount of time a program spends waiting for its I/O to complete. The time is measured from the point at which the program issues a knowledge-of-completion system call until the I/O completes. A program that completely overlaps I/O with computation is using the CPU while I/O proceeds. It will incur charges for CPU and I/O channel time but will have no I/O idle time. This encourages programs to initiate reading of data so that input is complete before the data are actually needed.

MEMORY FACTOR is currently 0.4 times the fraction of LCM used by the program. The fraction of LCM used is FLL + 230 (octal) divided by 400,000 (decimal). Since a program can vary FLL as it is running, the memory factor may also vary.

The Octopus system uses a different charging algorithm.

```
CHARGE TIME = (UNWEIGHTED CHARGE TIME)*(PRIORITY)

UNWEIGHTED CHARGE TIME = CPU CHARGE
     +(I/O FACTOR)*minimum(I/O TIME, BLOCKED TIME)

I/O TIME =
      QUEUE TIME
     +I/O CHANNEL TIME
     +VOLUNTARY LOAD/DUMP TIME)
```

QUEUE TIME is the time elapsed between the submission of your IOD to the system and the time it is given to the peripheral processing unit.

BLOCKED TIME is the time your code is in LCM but cannot use the CPU (usually because it is waiting for I/O).

I/O FACTOR on the Octopus machines depends on the time of day and on FLR, the fraction of LCM that you are using. During production periods, I/O FACTOR = (1/2) + (1/2)FLR. During the day on working days, I/O FACTOR = (1/4) + (3/4)FLR.

Charging algorithms can and do change. The ones given above were in effect

in August 1977. If you are concerned about minimizing your unweighted charge time, you should verify what algorithm is currently in effect.

FUNDAMENTALS OF CREATING AND EXECUTING CONTROLLEES

Introduction

The process of creating and executing a controllee is a mystery to most users. They follow recipes that work on simple programs and find that they generally work. For a production program, however, there is no excuse for such ignorance, because ignorance can lead you to do things wastefully. We shall present an overview of the process, which is described in greater detail by a whole host of other available documents.

Compiling

Normally, your programs or subroutines are written in FORTRAN or LRLTRAN. Compilation occurs when you or ORDER execute the CHATR (or CHAT) compiler with your source as input. Your source is a disk file with the text for your subroutines. The primary output of the compiler is a relocatable binary deck for each subroutine. Normally you never get card decks; instead, the images of the card decks are written one after another into a disk file. The compiler compiles the subroutines one at a time. Since the compiler is ignorant of where other subroutines and common blocks will eventually be in memory, it cannot generate instructions that reference the location of these externals. References to unknown addresses are flagged in the relocatable binary deck. Lists of needed externals are also included. Variables that are defined only within the subroutine itself are local variables. The compiler determines the length of the subroutine, including storage space for local variables. These local variables as well as internal locations to which the subroutine jumps are recorded in the relocatable binary as being a certain number of words away from the beginning of the program. Instructions referencing these locations must also be flagged so that the eventual memory addresses can be calculated and put in the instruction. A relocatable binary represents an intermediate state where most of the work of translating FORTRAN into 7600 machine instructions has been done, and enough information is available that the routines may be loaded anywhere in memory.

Normally the relocatable binary contains no information about internal variables and locations. That is, no correspondence is kept of a statement label and its eventual memory location. Nor are the locations of the local variables and their names included. This information is not necessary for running your code; but, if your code has bugs, it would be nice to know where everything went in memory. CHATR (and CHAT) can be asked to produce so-called type-26 cards or symbol-table cards in your relocatable binaries. The cards contain additional information about variables and labels and where they are

relative to the beginning of a subroutine or common block. These cards can be very useful in debugging a code, and you should read the CHATR (or CHAT) and/or ORDER manuals about getting these symbol-table cards.

Assembling

Occasionally you may need to write a code that is difficult to write in LRLTRAN or cannot be written efficiently in LRLTRAN. It may then be appropriate to write the program in an assembly language. Assembly languages are low-level languages in which you describe exactly which machine instructions you wish to use. Obviously you must know a great deal about the 7600 and interfaces to other subroutines to write in assembly language. (The NMFECC strongly urges its users to consult with it before embarking on assembly-language programming, so that it may fully apprise them of background information they need. If users must write in assembly language, the NMFECC suggests that users use COMPASS.) The principal output of an assembler is also a relocatable binary deck.

Relocatable Binary Libraries

A disk file containing images of relocatable binary decks can be loaded by the loader. The loader will run faster, however, if there is also a directory at the end of the file to describe the routines and tell where they are in the file. Public libraries such as ORDERLIB and STACKLIB have such directories. It is feasible and often desirable for users to create private libraries with directories. LIBMAK is the utility routine for creating and manipulating libraries. If your code is very long, you should save the binaries. When you repair a subroutine, you will then compile only the modified routine (compilation is an expensive process) and replace the relocatable binary in your private library. The library format (i.e., with directory constructed by LIBMAK) is preferred because it can be loaded more quickly.

Loading

In the simplest view of loading, you present one or more binary files (either relocatable binary libraries or the direct output of the compiler) to the loader and tell it at which subroutine the program is to start. The loader assigns memory locations to all the routines it needs, and all their common blocks. It then relocates all the codes, constructing an image on disk of the executable program. This disk image is the controllee. The loader constructs

the initial exchange package in the minus words and sets the P counter to the starting instruction location. The correct values of FLS and FLL are determined and written into words 16 (octal) and 17 (octal) of your program as well as into the exchange package. The loader also produces a map telling where all the subroutines and common blocks are located.

As a convenience to you, the loader assumes you want to start execution at a routine named MAIN. (the period is part of the name). Any routine that does not have a SUBROUTINE or FUNCTION declaration will be compiled into a program named MAIN. by CHATR (or CHAT). Even if your main program has a PROGRAM card that uses some other name, MAIN. is the name that will be assigned by CHATR (or CHAT). Thus, it is normally not necessary to tell LOD (the loader) where to start. Conversely, it is inappropriate to have more than one main program in your binary files.

If requested, the loader will also construct a symbol table, using any available type-26 cards. The symbol table can be thought of as an elaborate load map that tells not only where subroutines and common blocks have been placed but where internal variables and labels have gone. The symbol table may be placed in a separate disk file or may be in the controllee disk file following the controllee. The symbol table is intended for use by such debugging utility routines as DBUG, DBCTRL, and DOD.

As you recall, SSM is small. But, code can be executed only out of SSM; LCM can be used only for LCM common blocks. If you have a very large program, it may not be possible to load every subroutine into SSM simultaneously. You are then forced to use a technique called overlaying. For example, two sets of subroutines that are not needed simultaneously are grouped into separate code blocks. The loader writes both code blocks to the controllee file. The two code blocks are constructed in such a way that they must be read into the same SSM space, but only one can be in SSM and executing at any instant. A resident or level-1 code block must always be in SSM to control the reading of the overlays from the controllee disk into memory. As it turns out, you can have more than two overlays, and overlays can call in higher-level code blocks. In fact, LOD supports seven levels of code blocks, but you must be doing something wrong if you think you need more than three levels. If you use ORDER, you will see that it has control cards *MAIN, *OVERLAY, and *SEGMENT, which you can use to define three levels of code blocks. You will also have to become familiar with subroutine CHAIN in ORDERLIB, which you will normally use to ask for a code block to be read into SSM.

The overlaying process and other esoteric options are too much to be presented here. They are described in the LOD write-up, however, so we suggest you look there first and then (NMFECC users only, please!) consult with the NMFECC if you have problems. As an alternative to overlaying, you might also consider leaving all your data in disk files and running several controllees one after another to accomplish the same task as a large, overlaid controllee.

All controllees have an attribute named FLLCM. This is the initial load length or the number of words of the controllee, not counting the minus words that the system will read into LCM when the file is first executed. FLLCM is not written in the controllee; the system remembers it in a system table. LOD determines FLLCM and informs the system. Sometimes this number gets lost, such as when writing a controllee to FILEM and reading it back. In this case, FLLCM is zero, which means the entire file is to be read into LCM. This is usually harmless unless the file is too big to fit into LCM. Utility routines COPY and SWAT can be used to set the FLLCM of a file.

There are two other LOD options worth noticing. One eliminates the images of all LCM common blocks on disk. The second also does this and, in addition, eliminates numbered common in SSM if it is a single-code-block code and has numbered common last. For multi-level codes, this option eliminates the space on disk of that part of the SSM image that would have level-2 and higher-level blocks. There are two reasons for these options. One is that LCM common blocks cannot be data loaded (with DATA statements), and numbered common blocks in SSM should not be data loaded; therefore, since the disk space they would occupy has no information, it can be eliminated. The second reason is that higher-level code blocks are stored on disk between the initial core image and the symbol table if any. These loader options are often used to keep the size of the controllee file at a minimum. The loader will set the FLLCM to the compressed size. FLS and FLL in the exchange package will also be reduced; however, FLS and FLL in words 16 (octal) and 17 (octal) of your program will be the sizes actually required for execution. The first thing the program should do upon starting is to create a dropfile large enough to hold the fully expanded program. Then it should use the values of FLS and FLL in words 16 (octal) and 17 (octal) and ask the system to be allowed to expand to those amounts of space in memory. Both of these steps can be achieved easily by calling subroutine ADJUST in ORDERLIB, which issues the appropriate system calls. If your program is loaded under the control of ORDER, the compressing, dropping to another file, and expanding are done for you automatically, either by ORDER or by additional code that it adds to your program.

You should be aware that you are allowed to reset your FLS and FLL while you are running. Frequently, users place a single array in an LCM common block and position that common block at the end of their field length. Initially, the last word in the array (as determined by a DIMENSION) falls at the end of the field length (i.e., FLL). However, by creating a new dropfile and then increasing FLL (not exceeding the dropfile size minus 230 (octal) words), you can increase the size of the last array; that is, you can use subscripts greater than those declared at compile time. Adjusting FLL up and down is useful when various amounts of storage are needed at different times. The idea is to keep FLL as small as possible at all times because you are charged for the amount of LCM you use and because smaller jobs are more likely to get into memory than larger jobs at any given priority. Note, however, that adjusting FLS or FLL

always causes your code to drop to disk (a costly I/O operation); therefore, do not adjust field lengths unnecessarily.

Figure 1 shows how a controllee is laid out on disk. The symbol table is optional and need not be present. It may be placed in a separate disk file. The higher-level code blocks exist only for overlaid codes. Likewise, the space within FLS where higher-level blocks will be read for execution exists only if there are higher-level blocks. In Fig. 1, none of the loader options for squeezing out unused space has been used. The space for code blocks in FLS and common blocks in FLL is all zero. If you wish, you could use the controllee file as a dropfile, since it is large enough to hold the image of the executing program.

In Fig. 2, the unused space is eliminated. The compressed FLS and FLL shown are the ones in the exchange package. Your program can never reference addresses beyond these. The FLS and FLL you actually need are written into words 16 (octal) and 17 (octal) in the resident code block, and you must tell the system to expand your job in memory to these sizes. Also you will need a separate dropfile, since dropping into the controllee file (if it were large enough) would destroy your higher-level code blocks.

In the overlaying process, a fresh copy of the code block is always read in. Code blocks in SSM are never written back out to disk. If a code block is used a second time, there cannot be any variables in its subroutines that rely on being set from a previous execution. Information that must be preserved from one execution of a code block to a subsequent execution should be placed in a common block, and the common block should be forced into a lower-level code block.

A code block is a group of subroutines plus the common blocks they use that have not yet been defined at a lower level. Figure 3 shows how a code block is usually laid out. Subroutines with their internal variables come first, followed by all the labeled common blocks, followed by the numbered common blocks. On the load map, the name of a numbered common block is its number with two dollar signs appended. Blank common is named 999999$$. For a level-1 code block, you can data-load numbered common, but you shouldn't, because there is a loader option to eliminate the image (and hence the data) of the numbered common from the controllee disk file. There is another loader option used by ORDER that asks for code blocks to be constructed with labeled common first, followed by code, followed by numbered common. Under the default ordering as well as the ordering used by ORDER, numbered common is at the end of the code block. If the code block, furthermore, is at a higher level than 1, space will be allocated on disk only for the code and the labeled common blocks. Therefore, you cannot data-load numbered common in higher-level blocks.

As for LCM common blocks, the loader determines the amount of LCM space required by the various possible combinations of code blocks; that is, it determines how much LCM is required by the LCM common blocks in each combination of code blocks and includes it in FLL. As is the case with SSM common blocks, an LCM common block that is meant to be shared by two or more high-level code blocks must be declared in a lower-level code block through which the high-level

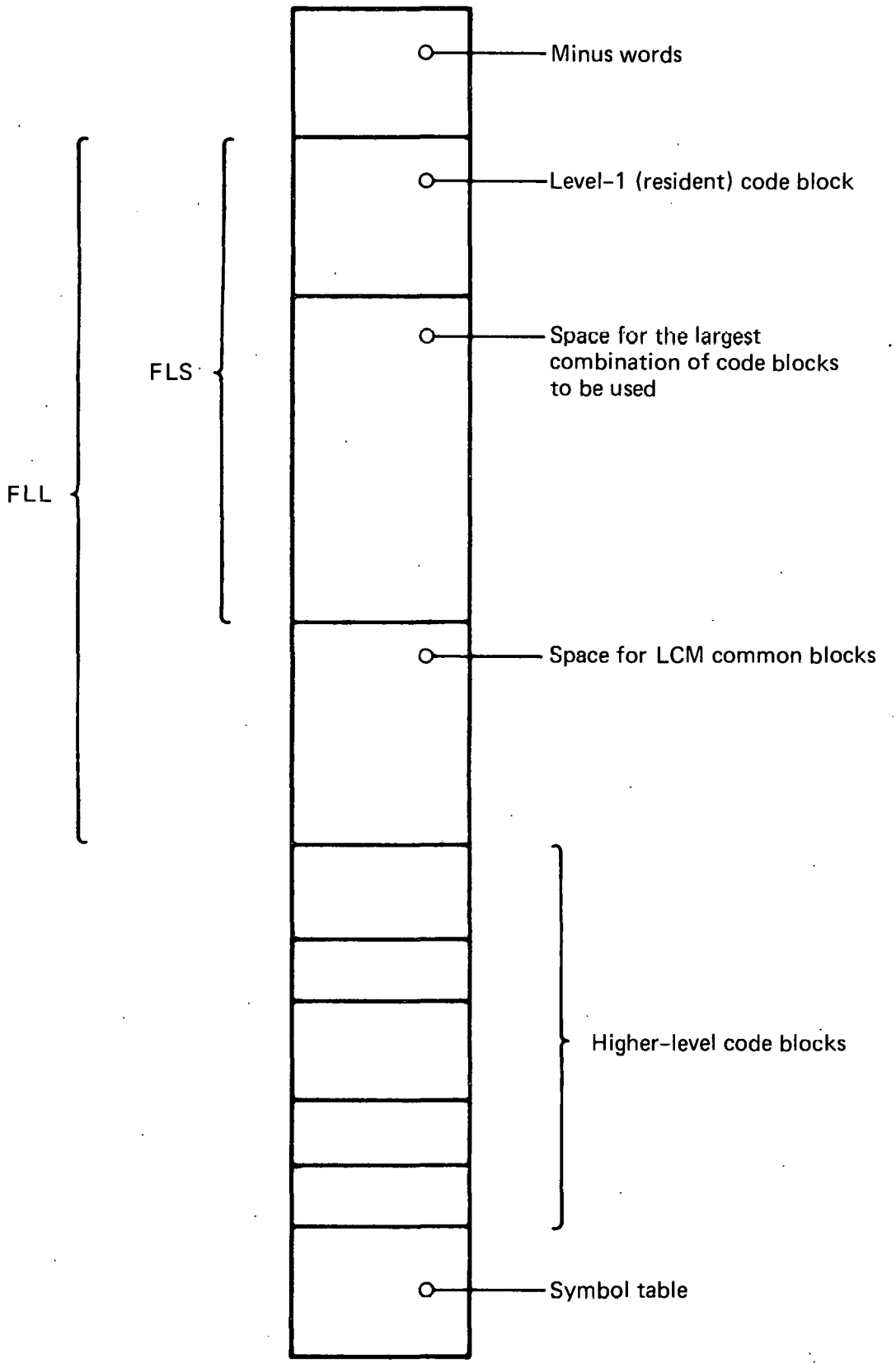


Fig. 1. Controllee layout on disk, where unused space is included.

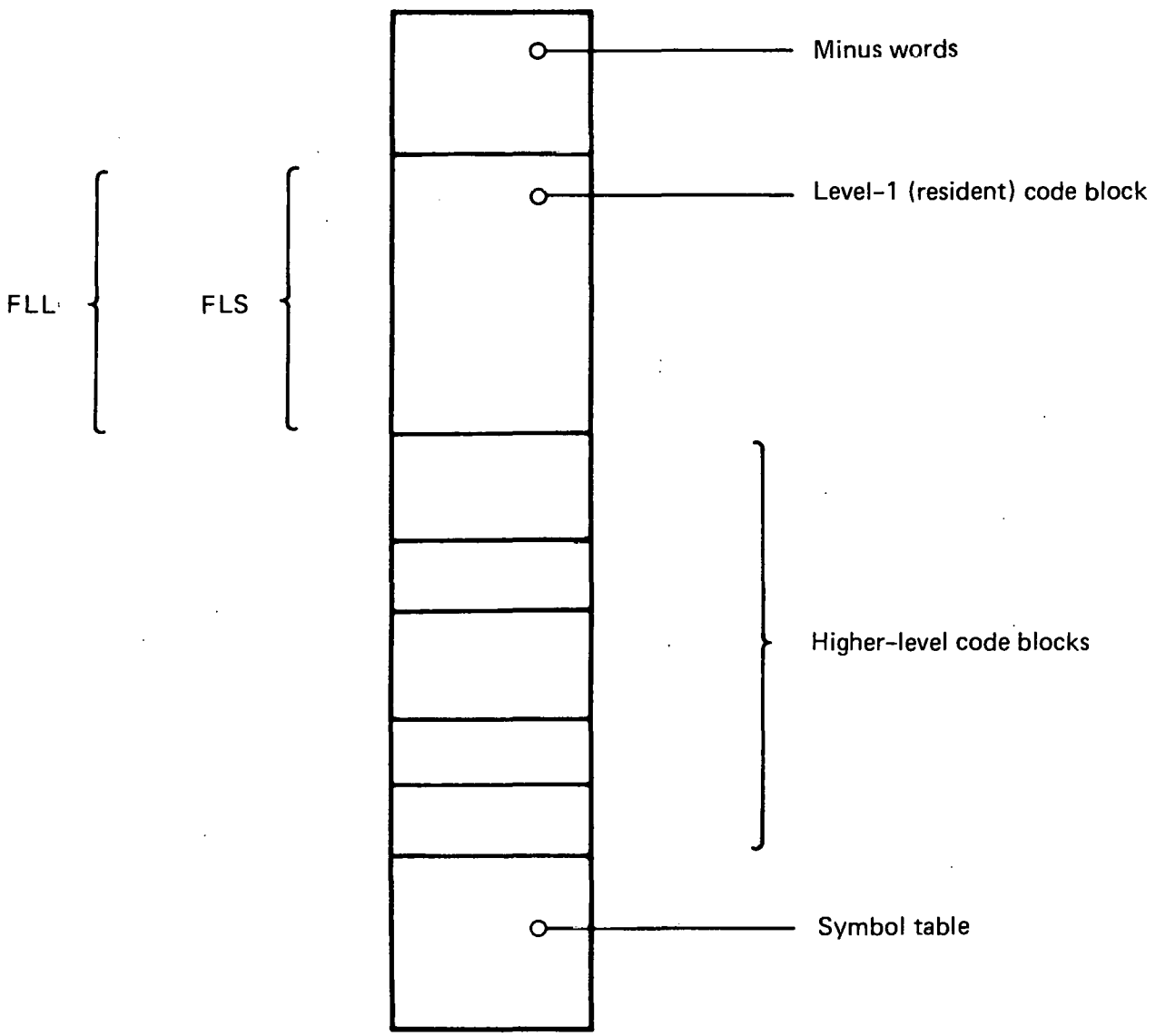


Fig. 2. Controllce layoul on disk, where unused space is eliminated.

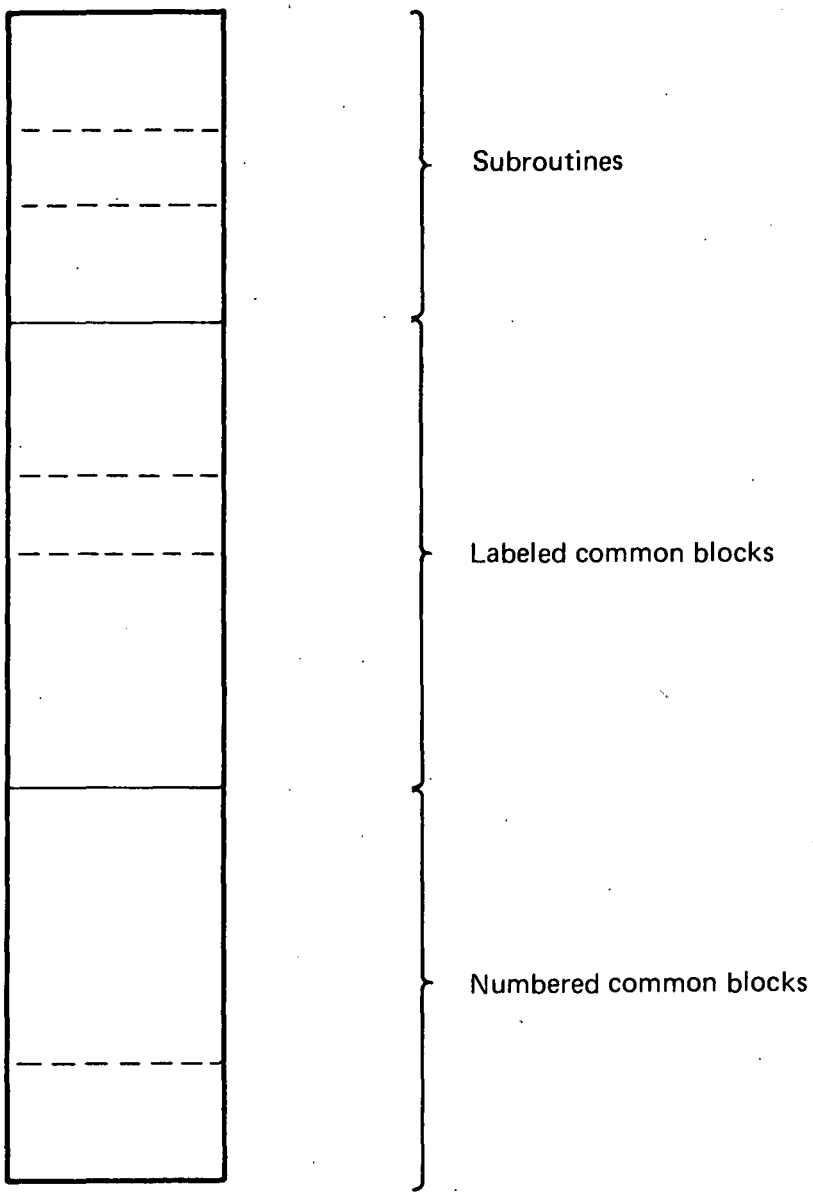


Fig. 3. Code-block layout.

code blocks are called. For example, two level-2 code blocks know nothing about each other's common blocks. If they are to share information, some subroutine at level-1 must declare the shared common blocks. That way the shared common blocks are allocated in the level-1 part of the LCM area. Otherwise each level-2 code block would have its own version of the shared common blocks somewhere in LCM, and the the locations of the shared blocks would in general not be the same; that is, the two code blocks have different ideas of where in LCM a given common block resides.

CONCLUSION

If all this seems somewhat complicated, we can assure you that it is! Large-scale scientific computing is not for casual programmers. Successful computing requires knowledge not only of the computational problem but of the tools available and how to use them. Suitably sobered, you should next read the treatise about preparing your first production code, in UCID-17557.

RGW/sv/lc

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		

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

Technical Information Department
LAWRENCE LIVERMORE LABORATORY
University of California | Livermore, California | 94550