GORDO MANUAL

VERSION 2.11

28 MARCH 1973

John Beatty
Charles Wetherell

PREFACE

This manual is intended for the use of moderately sophisticated systems and applications programmers who have an active interest in or are working on the Gordo system. Please use it in your work with the system so that it may be tested under fire. Please notify John Beatty (ext. 3006 or mail code L-63) if you discover inaccuracies, ommissions, or find something obscure.

In the body of this report, numbers which are specified in hexadecimal (base 16) will be contained in single quote marks, eg:

'1E000'

while numbers which are specified in decimal will be written in the normal fashion, eg:

8192

Bibleographic references are enclosed in parentheses, eg: (1). Footnotes are enclosed in square brackets, eg: [*].

A glossary of technical terms appropriate to the Gordo system may be found in Appendix 2 (page 88). The reader who does not understand the technical use of a word should look it up in the glossary. It is hoped that the glossary will enable people of varying backgrounds to effectively use this manual.

This document has been prepared with the TRIX report editing dialect "RED". It is hoped that the use of an interactive report editor will enable us to continuously and conveniently maintain the documentation for Gordo. The vast assistance of Hank Moll, author of TRIX and RED, is gratefully acknowledged.

A copy of this manual (including subsequent modifications) may be obtained from the photostore by executing

```
ELF RDS .055750:GORDO / I I
```

This file should be printed on the fr80, eg:

```
TRIX AC / I I
.RED
.PRINT(<XEROX GORDO BOX NNN GORDO MANUAL>)
.END
```

where "NNN" is your box number.

TABLE OF CONTENTS

INTRODUCTION

Gordo is an interaction-oriented time-shared computer system designed for experimental and production work in computer graphics and implemented on a Xerox Data Systems Sigma-7 computer. The system features six elaborate graphics consoles for the input and display of complex graphical and textual information. Gordo's design is biased towards those jobs which require a high degree of responsiveness and interaction on the part of the computer system. Only secondarily does it encourage the ordinary computational work of most computer systems. A primary goal of the Gordo project is research into the proper tools and techniques for computer manipulation and display of graphical information and into methods for man/machine communication via graphical devices.

THE GORDO HARDWARE

Gordo is implemented on a Xerox Data Systems Sigma-7 computer, a medium-scale, general-purpose, third-generation, 32-bit (8-bit byte) paged machine. The Sigma-7 was chosen for this project because it provides certain important hardware features useful in the construction of a time-sharing system. The hardware currently includes 32K 32-bit words of actual memory[*], a card reader, a card punch, a line printer, two 7-track magnetic tapes[**], an operator's teletype, a six million word swapping disk (also used for file storage), an Information International, Inc., display generator with four display consoles, and a Xerox Data Systems Sigma-3 computer, together with another display generator and two more consoles. The Sigma-7 executes all user programs and performs all time-sharing functions; the Sigma-3 acts only as a message display concentrator and display handler. Aside from the display generators and display consoles, the hardware is quite standard.

---

[*] A glossary of terms useful in a discussion of Gordo may be found in Appendix 1. The first use in this paper of terms which may be found in Appendix 1 is italicized. Also, within the glossary, the use of terms which may also be found in the glossary are italicized. The reader is assumed to have some sophistication in computer programming and to be familiar with the Sigma-7 hardware. A Sigma-7 machine manual is available.

[**] Even though the Sigma-7 has an 8-bit byte, 7-track tape units are used so as to be compatible with the Octopus network.

The display generators are digital devices operating out of the main memories of the Sigma-7 and the Sigma-3. The generators can display text, points, and lines at a number of intensities. Each generator may drive up to four consoles with any particular portion of the display list sent to any combination of the four consoles (although normally, each user would see only his own portion of the display). The display is refreshed from computer memory and a display genera tor is capable of maintaining a total of 600 full-screen vectors on the scopes to which it is attached without objectionable flicker. Each console consists of a keyboard, function box, light pen, and display screen.

Future hardware planned for the Gordo system includes an additional 96k words of core memory (for a total of 128k words), several small modifications to the display generators to allow a neater implementation of the systems graphics functions, and a connection to the Octopus system to allow Gordo access to the mass storage devices and worker computers of Octopus.

The Gordo system does not allow the normal user direct access to any of its hardware. Rather, the software which constitutes the operating system presents to each user a "virtual computer"[*] whose architecture is similar to but more hospitible than that of the actual Sigma-7 hardware. This virtual computer is connected to a console as its only input/output device. The user is thus is encouraged to concern himself only with the details of the virtual computer and the display generator and to avoid the vagaries of the actual hardware.

However, there are three aspects of the actual hardware which are of some interest to the user. The first of these is the size of actual memory available for use with user programs. This area, known as swapping space, is currently 22k words long. This size is not a limit on the size of user programs but is a useful guide to program design in some cases. The second important aspect is the size of the disk available to Gordo. The present disk holds 2624 512-word pages, of which approximately 500 are always occupied by system files. This limits the total amount of information which may be maintained within the system at any one time. The final important aspect of the hardware is simply that Gordo is implemented on a Sigma-7. This is reflected in the virtual computer architecture, which is similar to that of an actual Sigma-7.

---

[*] For a lucid description of the important concepts of "paging", "virtual machine", and "process" see [2], especially chapter 1. See also the applicable glossary entries.

A USER'S VIEW OF GORDO

As we have seen, A user of Gordo does not deal with a bare machine; system software intermediates between the user and the hardware. The user sees a combination of the two. This combination defines what is called a "virtual machine"; this is what the user sees from the vantage point of a user program. This section describes the virtual computer and some of its important features.

The user enters the system by logging-on. To do this, he must supply accounting and password information to the logger job responsible for limiting access to Gordo (details are described on page 14). Once over this hurdle, the logger creates a job for the user. Within Gordo, each user is uniquely and immutably identified by a job number which identifies the job created by the logger whenever that user logs into the system. Strictly speaking, the user is identified to the system by his badge number (and possibly his security combination), while the job created for him when he logs on is identified by his job number. But for all practical purposes the user and the job may be regarded as identical and both are identified by the job number. We note in passing that a user must be logged on while running a job.

The job created for the user may be regarded as a multi-processing computer network. Each job consists of one console and a number of processes, each of which controls a virtual computer. Initially, the job begins with just one process running, but during the course of computation the user may well cause the job to create and destroy a number of processes. These processes have completely independent facilities and are linked together only by the Gordo file structure and the fact that they are contained in the same job. The job may easily have several programs running in several different processes simultaneously, though more commonly only one process per job is actually active. Each job owns one console (with the exception of privileged systems jobs - in particular the Logger) and assigns that console to one of its processes. The user never communicates directly with the job, but only with its constituent processes. In fact, the job may be regarded as the glue holding a group of related processes together, with its current security level and job number the only features of direct interest to the user.

Jobs may create and destroy one another (though this power is not available to the normal user). The job may be likened to a large computer resource sharing network. Of course, Gordo only runs on one computer, the Sigma-7, and the description of jobs and the network of jobs is true in a logical sense only, not a physical one. It is the duty of the Gordo system to make this logical picture a reality. Some

details of that duty are discussed below.

THE VIRTUAL COMPUTER AND PROCESS

In spite of the longwinded discussion above, the Gordo user has a very simple machine to program. The virtual computer implemented by Gordo is probably easier to use than the real computer of most other systems. But what is a virtual computer?

The virtual computer is a facade presented to the user. We use the term "virtual" to indicate that a feature is actually provided by the software (sometimes with hardware assistance), and is not actually implemented in hardware. It is intended that the user write programs for the virtual machine as if it were, in fact, implemented in hardware. The hope is that the virtual machine provides a more hospitable and easily managed environment than exists on an "unassisted" machine.

Each virtual computer, as seen by a user of Gordo, is equipped with 128K words of virtual memory, 16 general purpose registers, the instruction set of the Sigma-7, and 8K of system code (B-system) loaded into the top of its virtual memory. The virtual computer is programmed just like a Sigma-7. A virtual computer may attempt to execute any instruction in the Sigma-7's instruction set. Some instructions (such as input/output instructions) will not be allowed as their execution by a user might violate the integrity of the virtual processor, but otherwise execution will have the same effect (except for some very arcane exceptions which are discussed in the machine reference manual and in Appendix 5) as on a bare Sigma-7. Requests for system services are made by means of subroutine calls into B-system. Among these calls are requests for file manipulation, graphics input and output service, and process control and manipulation. The traps provided for various unusual conditions are the same as those on the Sigma-7, but the trap information is returned in a slightly different manner.

It may seem that there is little purpose to the virtual computer, since it is so similar to the actual Sigma-7. But in fact there are two important reasons for its existence. First, it allows every user a memory space of 128K, rather than that of the actual computer, and provides a standardized, simple interface to the system. Secondly, the virtual computer is an object for which the operating system can more easily handle and allocate resources.

There are two important features of the virtual computer which differentiate it from an actual Sigma-7. The first of these is the display console. The display console forms a process' only direct link with the outside world (though there are indirect links through the file structure). Since Gordo serves as an interactive graphics facility, a great deal of programming is devoted to control of these consoles. The process communicates with the console through a number of B-system calls which read the keyboard, light pen, and function box, write the display screen, and change the console parameters. The average user will probably use one of a number of subroutine libraries to aid him in his programming. The console is in fact programmed much like a combination of a teletype and plotter and should not present any problems to most users.

The other unusual feature of the virtual computer is the relationship of the file structure to the virtual memory, which we shall now discuss in some detail.

THE FILE STRUCTURE

The two basic elements of the file system are "FILES" and "DIRECTORIES". A file consists of a system header page and some number (possibly zero) of 512-word data pages which comprise the body of the file and contain the data stored within it. The header contains pointers which specify the location of each page of information contained in the file, and records the file's security level, type of access (eg, read-only), and a reference count. Virtual memory is also partitioned in page-sized pieces called page-frames. File pages need not be accessed sequentially and any combination of pages in the file may be present or absent. Thus a file is a potentially very large unformatted randomly accessed congolmeration of data. File pages are referen ced sequentially by number, with the system header page starting the count at 0. A process may create and delete files at will.

A page frame in the virtual memory of a process is not usable unless it is associated with a page from some file. The act of forming this association is called "coupling" and is done by a B-system call whose arguments are the page frame number and the file page number; once associated, the file page and virtual page frame are said to be "coupled." Once coupled, a reference to any word in the page frame is a reference to the corresponding word in the file page (remember that pages and page frames are the same size). Any change in the page frame word is a change in the file page word. This is the key to file input/output under Gordo; once a file page is coupled into a page frame, it is referenced just like memory on an ordinary computer and may be

read or written by ordinary virtual computer instructions. Any given file page may be coupled into any number of page frames, possibly in different processes belonging to different jobs (which thereby share the file). A particular virtual page frame in a given process' virtual address space may have at most one file page coupled into it at any one time. Each file and each coupled file page may be associated with some access level which controls the uses to which the file page may be put. In particular, it is possible to restrict the ability of a process to write a file page. This provides for the use of shared data and program files without the risk of accidentally or maliciously modified code or data.

As an example, let us see how a program file is loaded into virtual memory and executed. The program is contained in a data file (programs are data, too). The file is opened and Master (itself a program, residing in a file) looks at page 1, which contains relocation information for the program. Master uses the relocation table to choose pages from the program file and to couple them into the correct virtual memory page frames. Once this is complete, Master transfers control to the start location of the program. Notice that the program is not copied into virtual memory, as might happen on some other systems. Instead, program file pages are simply associated with page frames. Once execution begins the program cannot distinguish virtual memory from actual Sigma-7 memory as long as it does not reference a virtual address not lying in one of the coupled virtual page frames. Also, program overlaying is done by simply coupling new program file pages on top of old page frames (automatically uncoupling the file pages previously in place).

A directory consists of a single disk page containing system header information together with some number of "directory entries". The directory heading, like the file heading, has access and security attributes and a reference count. In addition to access and security information, each entry has a lifetime, a time of creation, a name, and a pointer to the file header of the named file. A file is named only by a directory entry; a file or directory may have any number of entries pointing to it from other directories, possibly with different names. An entry may point to the directory in which it resides. A single directory may have two or more entries with identical file names. In searching such a directory, the most recently created entry having the given name will be the one found. The reference count of any file is a count of the number of places elsewhere in the system at which a pointer to the file may be found. Each time such a pointer is created, the reference count is increased; each time such a pointer is destroyed, the reference count is decremented. When the count drops to zero, the disk space allocated to the file is released. If the released file is a

directory, the reference count in each file for which there is a pointer in the released directory is decremented. Since files and directories themselves have no names they may be accessed only by the process which created them or symbolically through an entr y in another directory.

Disk storage for files is allocated in a straightforward manner. The disk is divided into pages and those pages not currently in use are kept on a free page chain. Whenever a new page is needed by a process (for a file page not previously referenced), it is obtained from the free chain. Whenever a file is finally discarded, the pages in it are returned to the free chain. When a file is first created, only its header page is actually allocated. New pages for data are allocated only as some process references the file page for the first time. This serves to minimize the number of disk pages actually allocated. This is desireable, since the disk is quite small. There is no way to return just one page from a file to the free chain. Programmers are encouraged not to reference pages until they are actually needed, and to delete their files when they log-off.

KEYWORDS

A process and a file are connected by a "keyword", which is similar to the I/O Connector (IOC) found in the minus words of a 7600 program. The keyword in which a file is "open" specifies an access level, a file type (directory or simple file), and the disk address at which the system header page for the file may be found. There are 16 keywords, of which 2 are reserved for system use. Four more are available to the user, but have conventional (and rarely altered) usages. A file must be associated with a keyword before its contents may be accessed.

When a file is created, it is created open in some keyword, and is referenced by specifying this keyword. There is no name whatsoever associated with the file, and it therefore may not be referenced by any process other than its creator, until it is entered from this keyword into some directory.

SHARING

INPUT / OUTPUT

The processes of a given job may read from and write to the keyboard and display console attached to the job; this is the only I/o device directly available to a user program. All peripheral I/O is

performed by a privileged system job called the I/O Job. There is one process in the I/O Job for each of the peripheral devices, namely the card reader, card punch, line printer, operator's teletype, and the magnetic tapes. With the exception of the card reader, the processes of the I/O Job have a special directory, called a mailbox, for each function (such as tape input).

A tape i/o request is made by creating a directory, into which is entered a pointer to each file for which input or output is desired. This directory is then entered into the appropriate mailbox. In the case of tape I/o, the name of this directory is the name of the tape to be requested. Thus to read the files "SXPL" and "XPL" from the tape "6-XPL10", the utility routine "TAPES" creates a directory "6-XPL10" and creates (empty) files named "SXPL" and "XPL". These files are then entered into the directory "6-XPL10". This directory is then entered into the mailbox "PUBLIC.INTAPE", where it is subsequently found and processed by the tape I/O process.

Printer and punch output are requested by entering the desired file into the mailbox for that process. The output will be identified with the file name and the job number of the user who created the file.

The card reader is treated somewhat differently. Card decks read from the card reader are placed in the directory "PUBLIC.GENERAL", from which they may be read by means of the Master "FETCH" command (see page 19). The id-card which precedes each card deck must specify a user; only that user may retrieve that deck from "GENERAL".

The I/O Job is functionally quite similar to user-1 on the 6600 and 7600 machines. Only the I/O Job may operate peripheral devices.

The Master "PRINT", "PUNCH", and "FETCH" commands, together with the "Tapes" processor, satisfy most file input and output requirements. It is quite possible, however, to write programs which prepare and mail file I/O requests directly. This is, of course, exactly what the system processors "Master" and "Tapes" do.

PROCESS SCHEDULING

It is, of course, true that Gordo is not implemented on the many computers which would be necessary if every process actually had its own computer. Instead, it is implemented on just one actual computer and the system allocates its resources to present the illusion that each user has his own computer (or computers). The three most important resources shared are CPU computation time, actual Sigma-7 memory, and

disk space for file storage.

Basically, computation time is shared equally among all processes. If there are N processes running each process is given 1/N of the total computation time available. This scheme is modified somewhat by a set of priorities. Each process is always listed on one of five scheduling queues. The priority of the queue as well as the position of a process within a queue determines who will run next. The five queues are the 'l interactive queue, the short quantum queue, the long quantum queue, the timer queue, and the input/output queue. Processes which have just received input from the console are placed on the interactive queue, the first to be serviced. When the process has run one short quantum time on the interactive queue, it is moved down to the short quantum queue. After another short quantum time, it is moved down to the long quantum queue. Once on the long quantum queue, a process round-robins with all other processes on that queue. Any console input read by the process will raise it back up to the interactive queue.. Since all processes on the interactive queue run before those on the short quantum queue, and those on the short quantum queue before those on the long quantum queue, this priority scheme rewards processes which regularly receive input from the console and which can process it quickly (in other words, it rewards those p.ocesses which are interactive). A process with only computation to do will fall to the long quantum queue and will not interfere with exclusively interactive users. Note that console ouput does not reward its process with increased priority.

The timer queue is for processes which have suspended computation for some reason until a specified time. Until then, they will not be scheduled for computation time. The input/output queue is a special one for those processes which service the peripheral input output devices. Processes on the input/output queue have priority over all others, since interrupts from peripheral devices must be handled very quickly. The actual computation time taken by these processes is not large.

The allocation of actual memory is done differently on Gordo than it is on most systems. Other systems try to bring all of the data for a process into memory at one time. But under Gordo there is only 22K of actual memory available for programs to use (presently) and a virtual computer may use all of its 128K of virtual memory. Clearly it is often impossible for all of a process' data to be in actual memory at the same time. So Gordo only brings into actual memory as much of the process' data as there is currently room for and then begins to run the process. If the process tries to reference a file page which is still on the disk, Gordo stops the process' computation momentarily and fetches the page from the disk into actual memory (possibly kicking some other file page back to the disk). All of this is invisible to the process and to

the virtual computer. However, such page faults do slow down the computation of the process in real time and cost the system considerable effort. A process which can run using only a few pages at each stage of its computation will get more work done in the same amount of real time. Programs which can share pages will each get more done, since the pages they need to run will often have been brought into memory by the other program. Finally, pages which have not been modified do not have to be sent back to the disk, since the disk copy is still correct. A page faulting process may also have to wait for pages to go back to disk to make room for pages coming into actual memory. Thus programs which can limit their writing to a few pages will fare better in real time. All of these effects should be considered when programming for Gordo.

The system does try to help processes in one way: a record is kept of the pages used during a shot. When the process is scheduled to be run again, the pages most recently used will be brought into memory first. (This is actually done while the previously scheduled process is executing.) Thus the process will find approximately the same pages in actual memory as when it last ran. For this to be useful, a program must refer to only a small number of the pages in its virtual address space over one shot time (currently '200' milliseconds). With the current hardware configuration, this pre-loading technique is not very effective, but it will sharply increase the system's effectiveness when additional actual memory becomes available.

OVERALL DESIGN OF THE SYSTEM

Gordo is composed of the following pieces:

1. A-system - the resident operating system.
2. B-system - the non-resident (paged or virtual) operating system.
3. Logger, I/O Job, and Garbage Collector - privileged system jobs.
4. Master - the system command interpreter.
5. System-users - privileged users.
6. Ordinary user jobs.

A-SYSTEM

A-system is responsible for basic system operation and control. Among its duties are job scheduling, disk scheduling, resource allocation, maintenance of communication with the display generators and consoles, handling of all interrupts and unusual machine conditions, execution of all input/output instructions, and maintenance of the virtual machine facades. A-system is basically concerned with the details of the actual hardware. The user is not aware of the presence of A-system and can request service of it only indirectly through B-system. A-system currently uses approximately 10K of actual memory, of which 4K is devoted to memory buffering for the local display generator.

B-SYSTEM

B-system is the user's interface to Gordo. Any system information which is available to the user, and all system services, are requested by calling B-system. B-system is re-entrant and actually resides in each user's virtual memory as a part of the user's program. B-system calls are implemented as subroutine jumps into the B-system space. B-system code differs from ordinary user code only in that it may call A-system, and may modify the user's state vector in non-trivial ways. One copy of B-system is shared among all users and the Gordo scheduling algorithm guarantees that only as much of B-system is resident in actual memory as is actually needed by some job.

PRIVILEGED SYSTEM JOBS

Generally speaking, there is one job for each user. Each job is assigned a separate console, and exists only as long as the user is logged into the system at that console. "Privileged jobs" are the only exception to this rule. They need not be attached to a console, and are

never killed.

The logger job is responsible for maintenance of the consoles when no user is making use of them. The logger keeps a log-on request displayed and transacts with the user attempting to enter the system. When a user presents a log-on request, the logger must determine that the user has legitimate access to Gordo, initiate accounting for the user, and then initialize a job for the user. When the user logs off again, the logger terminates the job, completes the accounting, and then clears the screen of all user display and returns to the log-on display.

The I/O Job performs all peripheral input/output tasks. Files are transferred to and from user processes and the peripheral equipment by placing requests with the input/output job. The input/output job sets up the appropriate commands to the input/output hardware and then requests A-system to execute those commands (remember that only A-system can execute input/output instructions). A-system also passes all interrupt information from the peripheral equipment to the input/output job. Note that the disk is regarded not as a peripheral device but as an extension of memory and that all transactions with the disk are handled directly by A-system. Input/output to and from the consoles can be handled by user processes with the intervention of B-system.

The Garbage Collector is responsible for conserving disk space. It periodically sweeps through the users' file structures looking for file entries which have exceeded their lifetimes. All such entries are deleted from the system. When all entries referring to a given file have been deleted, and if the file is not in use by any process (a file may be in use even if it is not entered in any directory), then the file in question can no longer be accessed in any way, and the disk space it occupies can be (and is) released for re-use. The garbage collector will never destroy a file actively in use. The normal user will never have direct contact with the garbage collector.

MASTER

The Master, like B-system, resides in the user's virtual address space. Specifically, the addresses between '1D000' and '1DFFF' are reserved for the Master. It loads and executes programs at the request of the user, and performs utility functions such as listing the user's root-directory, deleting files, sending files to be printed, etc. It also contains the XPL submonitor, which is the interface between XPL programs and the system.

SYSTEM-USERS

A system-user appears to the system as an ordinary user who has, however, the ability to issue privileged B-system calls. System-user status is available to systems programmers only, and is used for online maintanence of Gordo.

USING GORDO

In this section we will describe how one logs onto the system, enters programs and data into the system, and obtains output. In separate sections we shall also describe how one uses each of the major Gordo processes, such as the FORTRAN and XPL compilers.

LOGGING ON

A user who wishes to log on to Gordo must have a "user name" and an account number. In the past any string of ten or fewer characters (without contained blanks) was acceptable as a user name; future users will be identified to the system by their badge numbers for purposes of compatibility with the octopus network. People who wish to have their badge number and account entered into the system so that they may use Gordo should contact John Beatty or Gain Wong.

At the time a user's badge number is entered into the system he is assigned a unique job number. This job number is used to identify files, card decks, and listings.

If a console is not in use it will display the message:

LOG ON...<NAME>,<ACC NUM>,<SEC LEV>,<SEC COMB>

Only a name and account number need be supplied, unless the user has requested that he be issued a security combination. If the security level is not specified, it will have the default value of 2 (PARD). Typically, a user might type the following to log on:

803800,144PES

If the log on is successful, the "log on" message will disappear and the daily news will be displayed. When the news has been read, a carriage return will clear it and pass control to the Gordo command interpreter Master (described below). Master will display the message

MASTER PROCESS IN CONTROL

At this point the user is completely logged on and free to run his programs.

USING THE CARD READER

Decks are placed in the card reader face down with the 9-edge away from the user. Hit the "OPER'L POWER" button, then the "FAULT RESET" button, then "START." When the deck has been read through, hit "FAULT RESET" again, and finally "OPER'L POWER" again to turn off the power. The "STOP" button allows you to temporarily suspend the card reader while removing cards from the output hopper or placing additional cards in the input hopper.

Each deck is prefaced by an "ID card." If the ID card is not correct, the deck will be read through the card reader at a slower than normal speed. The ID card has the format

```
2 4 6 8(1)2 4 6 8(2)2 4 6 8(3)2 4 6 8(4)2 4 6 8(5)2 4 6 8(6)2 4
Idx          <name>,<job num>,<access>,<security>,<single user>
```

"X" may have the following values:

"A" read in a deck punched with EBCDIC character codes.

"B" to read in decks punched on Laboratory keypunches.

"C" to read in compressed decks (produced by the Master command PUNCH acting on a text file.

" " (ie, <blank>) to read in binary decks.

<name> is the file name. Note that it begins in column 15, and that subsequent fields follow immediately, comma delimited. <job num> is the job number assigned when your badge number was added to the Gordo Logger tables. The remaining three fields will be defaulted to 3 (read/write/execute access), 2 (for PARD security level), and 0 (not a single user file) if they are not present (see the respective glossary entries).

USING THE DISPLAY CONSOLES

All input to programs must be terminated with a carriage return. A running program may be broken by typing "CTRL-*". (any control character is typed by holding down the control key marked CTRL and typing the character in question.) CTRL-* is a program break and need not be followed by a carriage return. The Master will respond with a message indicating the program address of the instruction being executed at the time of the break. If a program terminates abnormally, Master

will display an "ABORT" message. When typing input, single characters may be erased with the backspace key. The whole line may be erased with the "CTRL-." key. If there are tab stops (which appear at the bottom of the screen, below the line on which typed characters appear), the tab key may be used to jump to the next tab setting. No input character is seen by any program under Gordo until the carriage return is struck to end the line. Once struck, the line is beyond recall. The next line may then be typed, but the carriage return will be ineffectual until the running program actually reads the previous line. Lines are echoed at the bottom of the screen as they are typed. Once the carriage return is struck they vanish from the echo area. Most programs will copy the line read onto some other part of the screen. New lines are added at the middle of the screen and old ones move off the top of the screen as the screen fills with output.

There are two other "control characters": typing "CTRL-+" will cause the current process to be killed. If the current process is the Master Process, control is returned to the Logger for the console, thus effecting a log-off. For any other process, the Master Process for the job is awakened. Typing a "CTRL-#" results in a log-off.

MISCELLANEOUS

It is important to note that files are shared and not copied in the Gordo system. If the user does a compilation, producing the listing file "RALPH" and passes that file to the printer for output, the file is only shared. If a new compilation is started, using the same listing file name "RALPH" the file being printed may be overwritten before printing is complete; the printer will be listing "RALPH" as the compiler writes a new listing on top of the old. To avoid this difficulty, delete files from your root directory as soon as the y are unnecesssary. This is a virtuous thing to do, as it conserves disk space.

It is possible to have several files of the same name in the root directory. A directory works like a stack and the last file of a given name which is entered will be the first one found for future use. When a DELETE is performed, the last-entered file of a given name will be the one deleted. On the other hand, the processors XPL-COMP, SLRK, and EDIT (as well as XPL object programs) will not create new files if they can find old ones with the required name.

MASTER

Master provides the means by which the user, seated at a display console, interacts with the system. Master executes commands which manipulate files and execute programs. It provides facilities for online debugging (for the more advanced user). It is automatically coupled into the virtual address space of the Master Process created by the logger at log-on time, and resides between '1D000' and '1DFFF'. Master similarly couples itself into process 2, which it forks when actually running a user program. Unless the user explicitly arranges otherwise, the Master is therefore a part of the virtual address space of each of his programs. An executing program may make requests of the Master by means of the B-system SERVICE call.

The commands which Master recognizes are given in the following table. Optional parameters are surrounded by square brackets, as "[...]". If a parameter begins with "file", it is to be a file name specified by the user. Parameters labeled "directory" similarly name a directory specified by the user. A file or directory name is a string of up to ten characters delimited by blanks, commas, semi-colons, or periods (depending on the context in which it is used). If any of these characters are to appear in the file name, they must be escaped. That is to say, they must be preceeded by a "CTRL-H". A "directory" may be a directory name, or it may specify a sequence of directory names, separated by periods, which specifies the access path by which a directory may be reached. Thus

PUBLIC.POSTOFFICE.GENERAL

names the directory "GENERAL", which is entered in the directory "POSTOFFICE", which is entered in the directory "PUBLIC", which is itself an entry in the user's root directory. Similarly, a "file" parameter may be a file name, or may specify the access pa th to a file. Thus

PUBLIC.POSTOFFICE.GENERAL.STRDET

names the file "STRDET" which is entered in the directory "GENERAL", reached by the access path described above. In the degenerate case,

LALR

might name a directory or data file which was entered in the user's root directory. Master will look first in the user's root directory for a file or directory. If it is unsuccessful, it will then search the shared directory "PUBLIC" which the Logger places in the user's root at

log-on time. A parameter named "loc" is a hexadecimal address. "Hex#" should be taken to represent any hexadecimal number. "Root directory" refers to the user's root directory (see the glossary).

BREAK [hex#,loc] "Loc" is the location at which the breakpoint named "hex#" is to be placed. "Hex#" is constrained to be a number between '0' and 'f', inclusive. If "loc" is ommitted, the old location of breakpoint "hex#" is kept and displayed. If both arguments are omitted, all breakpoints are displayed.

CALL name args This provides a method of issuing B-system calls from the console. "Name" is the name of a B-system call. "Args" is a list of the numeric arguments required by the system call, in the order given by the description of the call in Appendix B. For example, "OPEN" may be used to open a directory in some keyword of the process of which the executing Master is a part.

CLEAR [hex#, hex#...] The listed breakpoints are cleared. If no argument is present, all breakpoints are cleared.

DELETE file1 [file2 ... filen] Delete files "file1," ["file2," ... "filen"] from the root directory. Stop as soon as a file name cannot be found.

DESTROY Destroy all files in the root directory. This command should always be issued before logging off the system, as disk space is limited. (DESTROY actually deletes every entry in the user's root directory except "USERCOMPF" and "PUBLIC".)

ENTER directory file [name] enters the file identified oy "file" into the directory identified by "directory". If the third argument is present then the file is entered under that name.

EXECUTE file1	Execute the object program residing in the file "file1", which is entered in the root directory.
EXIT	See the command X.
FETCH file1 [file2]	Fetch the file named "file1" from the system directory "GENERAL" and put it in the user's private root directory. If "file2" occurs, the file is listed as "file1" in GENERAL but is to be entered in the root directory as "file2." Note that when a file is read through the card-reader the system enters it in GENERAL.
FILES [file1]	List on the screen the files currently in the root directory. If "file1" occurs, list instead the files in directory "file1," which is entered in the root directory.
GIVE file1 [file2]	Enter the file which appears in the root directory as "file1" in the system directory GENERAL. If "file2" appears, enter "file1" in GENERAL as "file2."
GO [loc1]	Continue execution of the last interupted (or loaded) program. This may not be successful if other Master commands have been executed since the interrupt. If "LOC1" is present execution continues from the designated location. Otherwise execution continues from the point of interuption. If the first instruction to be executed is a breakpoint, execution is suspended.
INSPECT loc1, loc2 ...	The contents of the virtual addresses specified are displayed.
LIST loc1 [loc2]	Display the contents of virtual address "loc1" in this process. If "loc2" appears, display the contents of virtual memory between "loc1" and "loc2," inclusive.
LOAD file	The file identified by "file" is loaded (coupled into the current processes virtual address space) without execution.

Breakpoints may then be inserted with the "BREAK" command, and the program executed by typing "GO".

PRINT file1 [file2] Print the file which is entered in the root directory with the name "file1" on the on-line printer. If "file2" occurs the printed output will be identified with the name "file2."

PROCEED [LOC1] Restart the last interrupted program. This command has exactly the same effect as "GO", except that the first instruction is executed, even if a breakpoint has been set at that location.

PSD hex# The program status doubleword ("PSD") of the last interupted (or loaded) program is set to the specified value. Note that this allows the condition codes to be set. If no argument is given then the psd is displayed.

PUNCH file1 [file2] Punch a card deck on the on-line punch for the file identified by "file1". If "file2" occurs the punched output will be identified with the name "file2." Text files will be punched in compressed text format. Binary files (output from Lfortran, SYMBOL, and DLOADX) will be punched as a sequence of binary cards. Binary files (described elsewhere) consist of card images, which this command punches.

PUNCHL file1 [file2] This command has the same effect as PUNCH except that text files will not be punched in compressed text format; ie, each text line will be transcribed without alteration onto a punched card, so that it may, if desired, be read into another machine.

REPLACE loc, hex#, hex#, ... beginning at the hexedecimal address "loc", the contents of the current process' virtual address space is replaced with the specified values.

STEP hex# — Executes exactly the specified number of instructions.

VALUE hex# — The argument is printed as a hexadecimal number (it is intended that at some time in the future an arithmetic expression be allowed as an argument).

X — Terminate the current process. To log-off, the user must always type this command once. If the response is

PROCESS 02 KILLED

then this command must be issued a second time to kill the Master Process and complete the log-off.

ZAP — Clear the screen of all display.

EDIT

INTRODUCTION

The Gordo system processor "EDIT" is an interactive text editor utilizing a Sigma-7 display screen for output and the associated keyboard for input. Edit allows a user to both create and modify Gordo text files.

Unless the user plans to generate his own text files, their exact format is unimportant. It suffices to say that a text file is a standard representation, inside the computer, of the information you would normally associate with cards prepared on a keypunch. Standard Gordo text files are generated by the card reader, the FORTRAN and XPL compilers, the SYMBOL assembler, and by EDIT. Gordo text files may be punched by means of the Master commands "PUNCHL" (to obtain a conventional card deck) and "PUNCH" (to obtain a compressed deck); text files may be sent to the line printer with the command "PRINT".

EDIT is a very flexible editor, and the language used to communicate with it may at times seem complex or verbose. In most cases, however, infrequently used options have sensible default values, so that there are only a few basic variations which must be learned in order to edit text files with reasonable efficiency.

During the execution of EDIT the display screen is divided into three sections. In the middle of the screen there are two horizontal lines; error messages are displayed between them. The last few commands issued will be echoed on the bottom of the screen. If a command cannot be understood, it will be echoed up to the point of error and then a question mark will appear. The name of the file being modified (the "current file") is displayed in the upper right hand corner of the screen. At any given time some particular line of text in the current file is singled out as the "current line". Eleven lines of text, centered on the current line, are displayed in the upper half of the screen. The current line is displayed at a greater intensity. (If it cannot be distinguished, the intensity of the crt beam should be reduced. A knob labeled "INTENSITY" is provided on the lower right hand side of the display screen for this purpose.)

DEFINITIONS

We begin with the following conventions:

"FILE" Stands for any FILE name.

(ctrl-k) This is the character generated by depressing the key marked "CTRL" and simultaneously striking the key for the letter "K".

(ctrl-h) This is the character generated by depressing the key marked "CTRL" and simultaneously striking the key for the letter "H".

<CR/;> represents a command terminator - all input lines to EDIT end with a carriage return, which is produced by striking the "RETURN" key. Thus each command is normally terminated by a carriage return. Multiple commands may be placed on a single input line if they are separated by semi-colons (";"). Such multiple command input lines must, of course, also end with a carriage return.

We shall use this symbol to represent one or more blanks as typed with the space bar at the bottom of the keyboard. This symbol indicates that at least one blank must be typed. Blanks are otherwise optional between the parts of a command.

The basic command designators are identified by the following symbols:

A ..... "Append" or "add" text lines to a file

D ..... "Delete" (a specified line or lines)

I ..... "Insert" (before a given line)

L ..... "List" or "locate" (a specified line, lines, or file)

N ..... Turn the screen display of edited text back on - Note that any error will automatically cause this command to be executed.

O ..... Turn off the screen display of edited text

S ..... "Substitute" one string for another

T ..... Set the tab stops on the display screen

U ..... Unimplemented command

V ..... Unimplemented command

X ..... "Exit" from the editor

The current line pointer is moved by using the following kinds of symbols in conjunction with the basic EDIT commands in a manner which we shall shortly describe:

integer	The line within the file numbered by the given (unsigned) integ er. Thus "10" moves the current line pointer to the tenth line in the current file.
+ integer	Move the current line pointer "integer" lines down (towards the end) of the file
− integer	Move the current line pointer "integer" lines up (towards the beginning) of the file.
!	This symbol designates the last line of text in the file.
[string]	This designates the first line following the current line which contains an ocurrence of the specified string.
<label>	This designates the first line following the current line which contains the given string in a label context. That is, the given label string and a line matching it must begin with the same character. Also, within a matching line, the label string must be followed by something other than a letter or digit.

SIMPLE EDITING

Roughly speaking, a command consists of a command designator (A, D, I, L, X, O, N, or T), sometimes a file name, and one or more line addresses as identified by text, label, or line numbers. For example:

D"file"10 20	means to delete lines 10 through 20 (inclusive) from the file named file.
L"file"15	means display (list) line 15 of file "FILE".

The most elementry editing operations are those of creating a new file, substituting one string for another in an old file, and inserting new lines or deleting old lines from a previously existing file. We shall now describe in cookbook form how one performs these simple operations. Those readers who wish to use the full power of EDIT should read the detailed descriptions of the various commands which appear in later sections.

To begin with, one runs EDIT by typing

EXECUTE EDIT

To create a new file:

type:	A"file"	where "file" is the name of the file to be created - the current file becomes "FILE" Note that text will be added to the end of "FILE" if it already exists.
		every succeeding line typed on the keyboard will be placed in the "FILE". Each line of text is terminated with a carriage return (and not with a semi-colon, which is an end-of-command symbol only).
type:	(CTRL-K)	(type this by simultaneously depressing the key marked "CTRL" and typing "K".) This removes EDIT from the append mode; it is then ready to accept another command.

To modify an old file:

type:	I"file"	The current file becomes "FILE", on which EDIT is prepared to operate. The current line pointer is set to the first line of text in the file.
type:	L integer	the line whose number is "integer" becomes the current line, and is displayed in the upper portion of the display screen.
type:	L[string]	the first line following the current line which contains an instance of "string" becomes the current line and appears on the display screen
type:	L+integer	the line "integer" lines down (towards the end of) the file from the current line becomes the new current line and appears on the display screen
type:	L-integer	the line "integer" lines up (towards the beginning of) the file from the current line becomes the new current line and appears on the display screen
type:	D	deletes the current line
type:	I	all input from the keyboard is inserted into the file immediately above the current line until the next (CTRL-K) is typed.
type:	A	appends all input from the keyboard to the currently open file, following the last line of the file, until a (CTRL-K) is entered.
type:	S*replace*pattern*	wherever the string "pattern" appears in the current line it is replaced by the string "replace". Here we have used an asterisk as a delimiter; it may not appear anywhere in the strings "replace" or "pattern". However, any character may be used as a delimitor

instead of "*", as long as it does not appear in either of the "replace" or "pattern" strings.

type: X exit from EDIT

Once again we emphasize that while the above techniques are sufficient to accomplish simple editing tasks, more sophisticated methods, which we shall describe later, greatly increase the power of EDIT.

TEXT ADDRESSING

Any line of text may be specified by giving the number of that line. For example, in the file

```
INTEGER A, B
A = B
```

the line "A = B" is line 2. But suppose that we insert a line after the integer declaration to obtain the file

```
INTEGER A, B
DIMENSION C(10)
A = B
```

Line 2 still exists, as does the line "A = B", but the line "A = B" and the line numbered 2 no longer correspond to each other. Line numbers, therefore, are not associated with the contents of a line, but merely with the position of a line in a text file at a given time.

While there are methods for allowing a line number, once affixed to a line, to remain linked to that line and its contents, these methods are often unnatural and more trouble than they are worth. Edit is instead designed to facilitate "context addressing". That is, EDIT allows you to locate lines by specifying a string of characters which may appear anywhere in the line. EDIT will search your text file and stop either at the first occurence of the specified string, or when it has searched the entire file.

Text searches are specified as follows: [string] is used to indicate that the text is to be searched for the first occurence of the string delimited by the square brackets ("[" and "]"). If the string includes a right square bracket ("]"), then the bracket must be

preceeded by the escape character "(CTRL-H)". Otherwise the right bracket would terminate the string of characters for which the search is to be made. This is a special instance of a general rule: EDIT makes particular use of the characters """, "]", "~", (CTRL-K), and ">". Situations arise, however, in which you do not wish EDIT to use one of these characters in its special meaning. To do this, preceed the special character with the escape symbol "(CTRL-H)". (This implies that to embed (CTRL-H) within a string one must type "(CTRL-H)(CTRL-H)" for the same reason.)

Thus to search for the string "A = B", type [A = B]. To search for the string "A[1,2] = B", TYPE [A[1,2(CTRL-H)] = B]. Note that since EDIT treats everything between the brackets as a string, [a = b], [a = b], and [a = b] are not equivalent.

A search normally begins in the current line plus one. EDIT continues searching for the string until it has searched the entire file and failed to find the string, or until it comes to a line containing the string. This implies that the file is treated as if it were circular; the first line of the file is considered to follow the last line of the file.

Pattern searches may be combined with relative displacements of the current line pointer (either positive or negative) to address lines which might not be easily addressable themselves but which are near lines which can be easily specified. Thus, the construct [ABC]+2 is perfectly legal. The search for "ABC" is performed and, if it is successful, the current line pointer is positioned according to the displacement (+2) as measured from the line in which the search succeeded. Displacements may also be used alone, that is, without searches. For example, +4 means the current line plus 4. Such addresses are evaluated from left to right in a command. Two addresses in a field, such as

[BBBFF] +3

denote a number of lines of text, ranging from the line in which the search succeeds (in this case) to that line plus 3. Note that the space between the search pattern and displacement means that there are two addresses in the field, not one.

Placing an integer immediately before the "[" indicates that the search is to start instead with the line whose number is integer. for example:

`[A = B]` starts the search for "A = B" in the current line plus 1.

`25[A= =B]` starts the search for [A= =B] in line 25 - note that there cannot be a blank between "25" and "[A = B]." it is important to realize that "25[A = B]" addresses the first line after line 25 which contains the pattern "A = B", whereas "25 [A = B]" addresses the block of lines in the text file between line 25 and the first following line which contains an instance of "A = B".

`[J = 1][A = B]` searches for "J = 1" starting in the current line plus 1 and, if successful, searches for "A = B" beginning with the line in which "J = 1" was found. If the search for "J = 1" fails, no search for "A = B" is made.

Often a line which you want to address may not be completely distinguishable from other lines. For example:

```
      .
      .
      .
C
      I = 0
 15   CONTINUE
         I = I + 2
         Z(I) = X(I)
      IF (I .LE. 10)  GO TO 15
C
      I = 1
 20   CONTINUE
         I = I + 2
         WX(I) = 1
      IF (I .LE. 20)  GO TO 20
C
      .
      .
      .
```

If the current line is "I = 0", and you wish to change the line "I = I + 2" appearing after statement 20, you cannot simply type [I = I + 2] because there is such a line immediately after statement 15 which will be found first. The line you want to correct can be addressed relative to the statement labeled 20, which immediately preceedes the statement to be changed.

For this purpose EDIT provides a facility known as a "label search." The rules for a label search are the same as those for a context search as outlined above, except that, to be successful, the string for which a search is being conducted must

(a) begin a line of text

(b) be followed by a character other than a number or letter.

Requirement (a) implies that either the label must begin in column one or that the string "LABEL" must include all blanks on the line which ar e to the left of the label. A label search is requested by bracketing the string with angle brackets instead of square brackets. Thus in the above example we can locate the second statement "I = I + 2" by typing

```
L <20>+1
```

The special symbol "!" always designates the last line of the text file.

When an address cannot be evaluated, either because a search fails or because the value of the line number is greater than "!" or less than 1, the symbol "?" is printed after the address, and the remainder of the command is ignored.

COMMAND FORMAT

The general format of a command is:

DESIGNATOR "FILE1" ADR1#ADR2, "FILE2" ADR3#ADR4 <CR/;>

where "#" is used to denote a required blank, and <cr/;> denotes the "RETURN" key or ";". Note that a blank is required between addresses, but is optional elsewhere. The designator and the <cr/;> are required for all commands. File names may be omitted if the file being referenced is the current file. In many cases, other parts of commands may be omitted. Various addressing combinations may be meaningless for a particular command. These matters are discussed in the descriptions of the commands. In what follows, "field1" refers to

"FILE1" ADR1#ADR2

while "field2" refers to

"FILE2" ADR3#ADR4

A - APPEND

Append adds text to a file, and therefore needs to know two things: where to find the text to be added, and where to put this text.

"FILE1" is the name of the file to which the text is to be added, and is called the "sink file." ADR1 and ADR2 select the block of text after which the new material is to be added. Everything beyond ADR2 in the sink file is deleted, while everything up to and including the line designated by ADR2 is preserved. If ADR2 is not specified, the new text is added after adr1. If ADR1 is also missing, text is appended after the last line of the sink file.

"FILE2" is the name of the file from which the text will be taken, and is called the "source file". ADR1 and ADR2 denote the block of text which is to be copied from "FILE2" into "FILE1".

Text is appended from "FILE2" until:

(a) all text from adr3 to adr4 has been appended

(b) an unescaped (CTRL-K) is encountered

Thus (CTRL-K) unconditionally terminates the append and places the editor in the command mode (ie, ready to accept a new command).

After the append has been completed, the current line for "file1" is the last line appended, and the current line for "FILE2" is the last line read from "FILE2".

If the second field is omitted, the text source is assumed to be the keyboard. In this case, text is appended until an unescaped (CTRL-K) is encountered.

EXAMPLES:

A "SINKFILE" 1 15, "SOURCEFILE" 1 ! <cr/;>

This adds all of the text in "sourcefile", from the first line to the last line, to "SINKFILE". The text is placed after line 15 of "SINKFILE"; all previously existing text in "SINKFILE" from line 16 on is lost.

A "NEWFILE" <CR/;>

This will place text typed in from the keyboard after the last line in "NEWFILE". If "NEWFILE" does not exist, it will be created.

A"NEWFILE","OLDFILE" 1
<CR/;>

This will copy "oldfile" to the file "NEWFILE", CREATING "NEWFILE" if necessary. If "NEWFILE" already exists, the contents of "OLDFILE" will be appended after the last line of "NEWFILE". Recall that if "OLDFILE" contains an unescaped (CTRL-K) in some line the entire file will not be copied. (Copying files is discouraged on the Sigma-7 because disk space is very limited. It is wiser to save a file on tape than to copy it. Remember - if the disk becomes full,

everybody loses.)

D - DELETE

Delete removes lines ADR1 to ADR2 from "file1". If ADR2 is omitted, only one line, namely adr1, is deleted. If ADR1 is also omitted, the current line is deleted.

If the second field is present (as indicated by a comma after the first field), then "FILE2" denotes a save file. Deleted text is appended to "FILE2" after adr4. If adr4 is omitted, text is appended after adr3. If adr3 is omitted, text is appended after the last line of "FILE2". If "FILE2" does not already exist, EDIT will create a file by that name and place the saved text into it. If the second field is not present, all deleted text is lost.

After the command is executed, the current line of "file1" is the line after the last line deleted, or line 1 if the last line of the file was deleted. The current line of "FILE2" is the last line appended.

EXAMPLES:

D "FILE1" 2 12, "SAVETEXT" <CR/;>

This deletes lines 2 thru 12 in "FILE1" and places the deleted text after the last line of "SAVETEXT". If the file "SAVETEXT" does not exist it is created and the text is placed in it. The current line of "FILE1" is the line that was formerly line 13 (if line 12 was not the last line of "FILE1"). The old line 13 now has the new line number 2.

D "TEST" <CR/;>

This deletes the current line of "test". The deleted line is lost.

D "TEXTFILE" 5, <CR/;>

This deletes the fifth line of "TEXTFILE" and appends the deleted text after the last line of the file whose name last appeared in the second field, no matter what the command naming that file was. That is, assume the command sequence

```
A "NEW" , "OLDFILE" 1
D "TEXTFILE" 5,
```

This would copy "OLDFILE" to "NEW", delete the fifth line of "TEXTFILE", and append the deleted line to "OLDFILE" after the last line of that file.

I - INSERT

Insert adds text to "file1" by placing the text to be added before adr1. If ADR1 is omitted, text is placed before the current line. Text is obtained exactly as in the Append command. The current line for "file1" is the last line inserted, while the current line for "FILE2" is the last line read from that file.

EXAMPLES:

I "FILE1" 5 <CR/;>

This inserts text typed in from the keyboard before line 5 of "FILE1".

I "FILE1","OLD" 1 ! <CR/;>

This inserts all the text of "OLD" (assuming a (CTRL-K) was not encountered) before the current line of "FILE1".

L - LIST

List allows positioning of the current line pointer of either "file1" or "FILE2". There are three cases to consider:

L "FILE1" ADR1 <cr/;>

The current line of "file1" becomes the line specified by adr1. The display screen shows lines (adr1)-5 through (adr1)+5, with the current line (ADR1) at a greater intensity.

L , "FILE2" adr3 <cr/;>

The current line of "FILE2" becomes the line specified by adr3. The display screen shows lines (ADR3)-5 thru (ADR3)+5

L "FILE1" ADR1, "FILE2" adr3 <cr/;>

The current line of "file1" becomes adr1. The current line of "FILE2" becomes adr3. The display screen shows lines (adr3)-5 through (adr3)+5 of "file2". ADR2 and adr4 are ignored if present.

S - SUBSTITUTE

Substitute exchanges one string for another. The syntax is slightly different than for other commands. The general form is

S *REPLACEMENT*PATTERN* "FILE1" ADR1#ADR2 , "FILE2" <CR/;>

The replacement string is substituted for every occurence of the pattern string in file "FILE1" between the lines designated by ADR1 and ADR2 (inclusive). The current file becomes "FILE1". Each line in which a substitution is made is appended to "FILE2" before the substitution is performed. If ADR2 is missing then the replacement occurs only on line ADR1. If ADR1 is also missing then every occurence of "PATTERN" is replaced on the current line. If "FILE1" is missing the replacement is done in the current file. If field2 is null (ie, only the comma is present) the text is saved in the file last named in field2 of a command. If the second field is entirely missing no text is saved. The text designated by ADR1#ADR2 is scrolled on the screen as the search for "PATTERN" progresses. If this is a large block of text it is much faster to turn off the screen display by means of the "O" command. The display may later be turned on by means of the "N" command.

EXAMPLES:

S*DISK PAGE NUMBER*PAGE*

This replaces every occurence of the word "PAGE" in the current line by the phrase "DISK PAGE NUMBER".

S?STH?STW? 12

This substitutes every occurence of the pattern "STW" in line 12 by "STH".

S NSPNRCPM1$! ! <CR/;>

This replaces every occurence of "NRCPM1" in the current file with "NSP".

N - START DISPLAY

N requires only the command designator and a carriage return. If the screen display has been turned off by a "O" command, "N" will turn the display back on.

O - OFF DISPLAY

O requires only the command designator and a carriage return. After execution of this command, EDIT will cease placing edited text on the display screen until an "N" command is executed. (Note that commands are ALWAYS echoed on the bottom half of the screen. The "O" command does not affect this.) "O" is primarily useful when using the Substitute command on large blocks of text. Each line of text will be scrolled onto the display screen - a relatively slow and time consuming process. Edit will run much faster in these circumstances if the display of edited text is temporarily turned off. Subsequent execution of the "N" command will turn the display on again. The occurance of an error will automatically turn the display on.

X - EXIT

Exit requires only the command designator and a carriage return. Exit closes all open files, releases space the editor previously reserved as "scratch" space, and returns control to Master.

T - TABS

Tab stops are available on all Gordo display consoles. They are identical in operation to those found on standard typewriter keyboards. The tab stops are displayed immediately below the line on which input is echoed; each symbol on the tab line represents a tab stop. Each time the "TAB" key is struck blanks are added to the input line until the next tab stop is reached.

Tab settings are initialized when the system is loaded, at which time the rightmost tab symbol ("O") marks column 80. They may be reset by means of the EDIT "tab" command,(*) or by means of the

system processor "TABSET". Tabs remain set even after the user has exited from EDIT and logged off the machine. they remain at the same setting until explicitly altered by a B-system CONTROL CALL or a "tab command" in EDIT, possibly by a different user.

To set the tab stops:

type: EXE EDIT — Tabs may also be set by the processor TABSET, which affords more flexibility that the tabset command of EDIT. Tabs may in fact be set by any program, using the B-system "CONTROL" call.

type: T — the next input line will be copied into the tab buffer. Therefore, every non-blank symbol in the next input line will become a tab stop.

UNIMPLEMENTED COMMANDS

The command designators "U" and "V" correspond to unimplemented commands. Typing one of these will result in an error message, but the editor will continue normal execution. All slashes (/) appearing in command strings must be escaped (again, this corresponds to an unimplemented feature).

BUGS

It is not wise to use the same name for "file1" and "file2" since the editor does not always do the right things with its internal pointers to the file.

Using the delete command to remove more than one line at a time may occasionally cause extra text to be destroyed.

(*) The tab command has no effect on consoles attached to the Sigma-7 through the Sigma-3. The standard settings on the Sigma-3 are at columns 7 and 73. No tab symbol is displayed beneath the echo area at the bottom of the screen. While a line is being typed, tab skips will not be echoed on the screen. However, the line will be correctly entered into the file.

There are certain error messages that you should pay particular attention to. One of these is the "HELP" message. It is used, in general, to denote a place in the editor program that you should not have reached. It usually denotes a bug in the editor, and should be brought to the attention of John Beatty or Gain Wong. The other messages of interest are

BAD LINE IN FILE
MAP ERROR
FILE ERROR

Unless you are editing a file which was created or modified by a program other than SYMBOL, FORTRAN, XPL, EDIT, or the i/o devices, messages such as these indicate that the file has been damaged (eg, wa s not correctly read from tape). The message

NOT A TEXT FILE

means just what it says. If you read a card deck through the card reader intending to edit it and received the above message, chances are that you did not properly prepare the id card.

SAMPLE EDITING SESSION

SYMBOL

"SYMBOL" is a relatively unsophisticated assembly language supplied by XDS. A reference manual is available (18) which completely and definitively describes the language.

To run the SYMBOL assembler, type

EXECUTE SYMBOL

In response you will be asked to type a command string. which is of the following form:

<SFN>,<LFN>,<BFN>,T

where

<SFN> is the name of a text file in your root directory which contains the program to be assembled.

<LFN> Is the name of the file which will contain the listing generated by the assembler. If this file does not exist in the user's root directory it will be created. If a file by this name already exists in the user's root directory, the listing will be written on top of the previous contents of the file.

<BFN> is the name of the binary file into which the assembler writes reloc atable object code. If this file does not exist in the user's root directory it will be created. If a file by this name already exists in the user's root directory, generated code will be written on top of the previous contents of the file, which are thereby destroyed.

T indicates that error messages from the assembler are to be printed on the display screen as well as on the output listing.

If <LFN> is omitted, no listing file is generated. If <BFN> is omitted, no binary relocateable code file is generated. If "T" is omitted, error messages appear on the listing but are not displayed on the console. Thus the following are all valid command strings:

```
ASYS,ASYSLIST,ASYSBIN,T
ASYS,,ASYSBIN
ASYS,ASYSLIST
```

Upon conclusion of the assembly, another command will be requested. The following are acceptable:

L indicates that the listing file <LFN> is to be sent to the on-line printer. <LFN> will be deleted from the root directory. <LFN> may also be listed by means of the Master "PRINT" command, which does not delete the specified file.

P indicates that the relocateable binary file <BFN> is to be punched . <BFN> will be deleted from the root directory. <BFN> may also be punched by means of the Master "PUNCH" command, which does not delete the specified file.

R indicates that the assembler is to be re-initialized for a new, and entirely independent, assembly.

X causes the assembler to exit to the Master.

Before the assembled program can be executed it must be loaded, using the processor "DLOADX", and the resulting object file converted to the "old executable file format" by means of the processor "FLD". DLOADX and FLD are discussed separately.

For very large programs it may be necessary to enlarge the size of the symbol table. To do so, type

EXECUTE SYMBOL (N)

where 'N' is the hexadecimal number of pages to be allocated for the symbol table. The default value for 'N' is '10'.

FORTRAN

The FORTRAN compiler is an XDS supplied processor which is definitively described in the XDS FORTRAN reference manual (19). It generates inefficient and non-reentrant code. To invoke the FORTRAN compiler type

EXECUTE FORTRAN

The user is then asked to supply a command string, which is of the following form:

<SFN>,<LFN>,<BFN>,T,S,D

where

<SFN> is the name of a text file in your root directory which contains the FORTRAN program to be compiled.

<LFN> is the name of the file which will contain the compilation listing generated by the compiler. If this file does not exist in the user's root directory it will be created. If a file by this name already exists in the user's root directory, the listing will be written on top of the previous contents of the file. If <lfn> begins with an "L", the code which is generated for each FORTRAN statement will be listed immediately following the statement in the output listing. This typically increases the size of the listing file by a factor of 5.

<BFN> is the name of binary file into which the compiler writes relocateab le object code. If this file does not exist in the user's root directory it will be created. If a file by this name already exists in the user's root directory, generated code will be written on top of the previous contents of the file, which will thereby be destroyed.

T indicates that error messages from the compiler are to be printed on the display screen as well as on the output listing.

S Indicates that in-line SYMBOL assembly language code may appear within the body of a FORTRAN routine. (An "S" in column one of a line of the FORTRAN program indicates that the line contains a symbolic machine language instruction.)

D indicates that the compiler is to generate code to implement the FORTRAN IV-H DEBUG facilities discussed later in this section.

If <LFN> is omitted, no listing file is generated. If <BFN> is omitted, no binary relocateable code file is generated. If "T" is omitted, error messages appear on the listing but are not displayed on the console. If "S" is omitted, inline assembly code is not allowed and will result in a compilation error. If "D" is omitted, DEBUG code is not generated. The source, listing, and binary code file names, and the "T", "S", and "D" options must follow each other in the indicated order. Thus the following are all legal command strings:

```
SBOOT,BOOTLIST,BOOTBIN,T
TABSET,LTABSET,BTABSET,T,S,D
TABSET,,BTABSET,T,,D
SPLINE,,BSPLINE
```

Upon conclusion of the compilation, another command will be requested. The following are acceptable:

L indicates that the listing file <LFN> is to be sent to the on-line printer. <LFN> will be deleted from the root directory. <LFN> may also be listed by means of the Master "PRINT" command, which does not delete the specified file.

P Indicates that the relocateable binary file <BFN> is to be punched . <BFN> will be deleted from the root directory. <BFN> may also be punched by means of the Master "PUNCH" command, which does not delete the specified file.

R Indicates that the compiler is to be re-initialized for a new, and entirely independent, compilation.

X causes the compiler to exit to the Master.

Before the compiled program can be executed it must be loaded, using the processor "DLOADX", and the resulting object file converted to the "old executable file format" by means of the processor "FLD". DLOADX and FLD are discussed separately.

For very large programs it may be necessary to enlarge the size of the symbol table. To do so, type

EXECUTE FORTRAN (N)

where 'N' is the hexadecimal number of pages to be allocated for the symbol table. The default value for 'N' is '10'.

THE FORTRAN DEBUG PACKAGE

The FDP is a line-by-line debugger operating at the FORTRAN statement level. The decision to run under FDP is made at compile time and each separately compiled routine may be compiled with or without the addition of code to implement the debug package. Each routine uses the package independently of other routines, so that a running program may run under FDP at some times and free of it at others. At the time a FDP-compiled routine is entered, the programmer may choose one of several modes of operation for the rest of the routine. In the course of executing the routine, the debugger may supply the user with information about the logical flow of control within the program, or may display the values of variables whose contents change, and may accept a re-definition of the run mode.

In order to use the debug package on a routine, the "D" option must be specified at the time the routine is compiled. As an absolute minimum, the FORTRAN command string must contain a source file name as the first parameter, a binary file name as the third parameter, and "D" as the sixth parameter:

<SFN>,,<BFN>,,,D

The FORTRAN compiler assigns a line number to each line of the source file. These line numbers are used by the FDP to specify locations within the program.

To run the compiled program under FDP, CLOADX and FLD are used as usual and the FLD-ed file is executed in the normal fashion by means of the Master "EXECUTE" command. At the time a routine compiled with the "D" option is first entered, execution of the program will halt, the FDP will take control, and the name of the routine just entered will be displayed. The user may then select a run mode by means of the S, V, T, R, and A commands, and their complements NS, NV, NT, NR, and NA. The user may type any of the above commands whenever program execution has been suspended and FDP is expecting input. Execution of the suspended program is always resumed by typing the command "G". Striking the

"RETURN" key re-issues the last command. We shall now discuss each command in turn.

S, NS (Step, No-Step) - If S is typed, FDP will halt before executing each statement in the current routine, and will print the line number of the statement about to be executed. That is, S forces FDP to execute the routine statement by statement under user control. Typing "NS" will cancel step mode. Execution is resumed by issuing a "G" command.

V, NV (Variable display, No Variable display) - Any future assignment statement in the current routine will cause the name of the variable to which an assignment is made, and the value it receives, to be displayed. If the variable is subscripted, the linear subscript will be printed also. NV will cancel this mode.

T, NT (Trace, No-Trace) - Typing T places FDP in the trace mode. Until NT is typed, the line numbers of all executable statements in the current routine will be printed as the statements are executed. NT cancels this mode.

R, NR Normally, the name of a routine which was compiled with the "D" option and run under FDP is displayed when it is entered; the name of the calling procedure is normally displayed upon exiting the routine. Typing R will suppress the display of names at entry and exit, and eliminate the halt at entry. NR restores this feature.

A, NA (Address stop, No Address stop) - Typing "A<number>" will result in a halt immediately before executing the statement on line "<number>". Note that there are no blanks between the command designator "A" and the line number specified. Only one address stop (the last entered) is maintained in any one routine. Once placed, an address stop will remain until a new one is placed in the same routine, or until the stop is removed with the NA command.

The default settings are "NS", "NV", "NT", "NR", and "NA".

If the debug package does not understand a command, the message "EH... WHAT?" will be displayed.

THE FORTRAN LIBRARY

Fortran Library

DLOADX

The Dumping Loader (DLOADX) processes relocatable binary files created by both the SYMBOL and FORTRAN IV-H language processors. Its primary job is to resolve forward references, merge multiple binary files together, and to resolve references to the system library (see page 85).

A single relocatable binary file may be created from the card decks punched from the binary files produced by different assemplies or FORTRAN compilations by separating each deck with an !EOD card and terminating the deck with two !EOD cards.

To use DLOADX, type

EXECUTE DLOADX

You will then be asked to type a command string. There are two acceptable commands: "Load" and "Dump".

LOAD

The Load command has the form

L(<FNL>)(<LFNL>)B=<HEX#>,M,N

where

<FNL> is a list of file names, separated by commas. These are assumed to be relocatable binary files. Every routine in each file is loaded.

<LFNL> is a list of library file names, separated by commas. These files are searched for references which are unresolved after loading the files in <FNL>. Only those routines which are needed are loaded. This field is almost always superfluous, and is therefore generally omitted. In general, FORTRAN generated binary modules will not load properly if placed in a library file.

B=<HEX#> fixes the load bias in words. the files in <fnl> are to be relocated relative to the hexadecimal address specified. The default value is '200'.

M causes a load map to be written into a text file called "MAP," which is entered into the root directory.

N specifies that the system library ("SYSLIBX") is not to be searched for unsatisfied references. If this option is not included in the command, the library SYSLIBX will be searched for unsatisfied references.

The Load command causes DLOADX to: load the binary files specified by <FNL> into DLOADX's virtual address space; resolve any forward or external references; load explicitly referenced routines from the private library files <LFN> (if present); and search the system library for any remaining unresolved references.

DUMP

A Dump command string has the form

D(<FN>)<LAP>

where

<FN> Is the name of a binary file into which the contents of DLOADX's virtual address space is to be dumped. If this file does not already exist in the root directory, it is created. If it does already exist, the previous contents of the file are destroyed.

<LAP> Is a list of hexadecimal address pairs "<addr1>,<addr2>"; pairs also are separated by commas. For example:

100,300,450,1200

The virtual addresses from <ADDR1> to <ADDR2> are dumped, for all of the pairs specified. If <LAP> is empty (no address pairs are given), DLOADX will use an algorithm of its own to determine the portions of the address space which should be dumped. This is always sufficient, but may unnecessarily include large blocks of uninitialized data space. Use of <LAP> to achieve selective dumping may substantially reduce the size of the dump file <FN>, which is usually desireable if the file is to be punched.

The dumped file must be processed by FLD (described in a separate section) before it can be executed.

MODIFY

The Modify command has the form

M <NAME> <HEX#>,<VALUE>,<VALUE>,...

where

<NAME> is an external definition, that is, a symbol defined within the loaded process, and available for external linkage (ie, listed on the load map generated by Ldloadx).

<HEX#> Is a positive or negative hexadecimal value. If <NAME> appears, then <HEX#> is a displacement relative to the location <NAME>. If <NAME> does not appear, then <HEX#> is an absolute hexadecimal address. In either case the effect is to specify the hexadecimal address at which the loaded program's virtual address space is to be modified.

<VALUE> Is a hexadecimal value which is to be inserted at the location determined by <NAME> and <HEX#>. If more than one value is given, subsequent values will be inserted in successively higher, adjacent locations. When <HEX#> is used to specify an absolute hexadecimal location, the absolute hexadecimal location must contain a leading zero. Each <VALUE> is right justified in its respective word if it contains fewer than 8 digits.

The Modify command allows the user to modify the contents of a program which has been loaded into DLOADX's virtual address space.

DEFXREF (define external reference)

This command has the form

X <SYMBOL>,<VALUE>

where

<SYMBOL> is the name of the external reference to be defined.

<VALUE> is a hexadecimal value which is to be assigned to the indicated external definition. <VALUE> may itself be the name of another externally defined symbol, or the name of an externally defined symbol followed by a hexadecimal displacement, or an absolute hexadecimal address.

This command allows the user to define an external reference before loading a binary file. It must be issued before ANY other command.

DLOAD

In almost all cases user programs which require library subroutines are linked to the re-entrant library "SYSLIBX" which is shared by all users. DLOADX assumes this to be the desired effect. In a few cases, however, it may be desireable instead to load from the old, non-reentrant library. In this case each subroutine needed must be copied into the dump file <FN>. To achieve this, there is an alternate entry point to the dumping loader, namely "DLOAD". Executing DLOAD will result in referenced library routines being copied from the old system library "SYSLIB" into the dump file. However, as "SYSLIB" is no longer maintained in the directory PUBLIC, the user desiring to run DLOAD must arrange to obtain a copy of SYSLIB and place it in his root directory.

Please note that at some time in the future, to be announced, DLOAD will assume the functions of Dloadx and its current function will be eliminated.

EXAMPLES:

```
L(ASYSBIN)B=0
D(ASYSEXE)40,70,1200,2E00
```

The contents of the relocatable binary file "ASYSBIN" generated by the ass embler are loaded into DLOADX's virtual address space. The addresses between '40' and '70', and between '1200' and '2e00' of the load ed program are then dumped into the new relocatable binary file "ASYSEXE". As it happens, the program actually uses address between '200' and '1200' as w ell, but this block of memory is a data area whose contents are not initialized and therefore do not need to be dumped.

L(TABSETBIN)
D(TASSET)

The contents of the relocatable binary file "TABSETBIN" generated by the FORTRAN compiler are loaded into DLOADX's virtual address space. Forward references are filled in and references to routines in the system library "SYSLIBX" are resolved. The loaded program is then dumped into the binary file "TABSET".

L(BOOTBIN)(PRIVATELIB)M,N
D(BOOT)

The relocatable binary file "BOOTBIN" generated by the FORTRAN compiler is loaded into DLOADX's virtual address space. Forward references within BOOTBIN are filled in. The binary file "PRIVATELIB" is then searched for routines which are named but not defined in BOOTBIN. If the referenced routine is found, it is loaded. The "N" indicates that if there are external references which cannot be found in PRIVATELIB, they are to be left unresolved (that is, the system library is not to be searched in an effort to find the still unresolved references). The "M" indicates that a load map is to be created. It will be entered into the root directory with the name "MAP".

THE LOAD MAP

What follows is an example of a load map generated by DLOADX:

```
PREF            B
SREF            C
PREF            BF:PIN
PREF            EXIT
PREF            BF:SX
PREF            BF:SS
PREF            BF:SR
UDEF    60 0    M:C
UDEF    65 0    M:OC
UDEF    6A 0    M:LO
UDEF    6F 0    M:LL
UDEF    74 0    M:DO
UDEF    79 0    M:PO
UDEF    7E 0    M:BO
UDEF    83 0    M:LI
UDEF    88 0    M:SI
UDEF    8D 0    M:BI
```

```
        UDEF    200 0    A
        UDEF    200 0    LOWESTLOC
        UDEF    209 0    RALPH
        DEF     2D4 0    NAME
        DEF     2EA 0    IGRAPH
        UDEF    35A 0    HIGHESTLOC
        UDEF   2002 0    M:LORST
        UDEF  1799C 0    F4:COM
```

where

DEF "(used) definition" - thus the external symbol IGRAPH is defined and used, and represents byte 0 of word '2EA'.

UDEF "unsatisfied definition" - thus M:C has been defined, but was never used.

PREF "(unsatisfied) primary reference" - for example, BF:PIN was used, but not defined, even after having searched the system library.

SREF "(unsatisfied) secondary reference" - this indicates the the corresponding external symbol was referenced, but not defined outside the system library (that is, the system library was not searched for the symbol, which was undefined).

FLD

"FLD" is a system processor whose function is to convert loaded binary files generated by DLOADX into the "old executable file" format (see page 71). To run FLD type

EXECUTE FLD <FILENAME>

where <FILENAME> is the name of the file in the root directory which is to be converted to the old executable file format. The resulting file may be directly executed by typing

EXECUTE <FILENAME>

It is sometimes useful to know that FLD creates a new file with the name <FILENAME>, into which the executable code is written. Thus after issuing the MASTER commands

```
GIVE TAPES BTAPES
EXECUTE FLD TAPES #IN 15 FETCH BTAPES
```

the root directory will contain an entry for the file BTAPES, which was created by DLOADX, and for TAPES, which is the executable file created by FLD. BTAPES can then be FLD-ed again at a later time to obtain a new copy of the executable file. If one does not thus explicitly preserve a pointer to the relocatable binary file created by DLOADX, it will be deleted and its disk pages released when FLD is finished.

PEEK

"PEEK" is a system processor whose function is to monitor the number of disk pages in use. To run peek type

EXECUTE PEEK

A graph will appear on the display screen whose x-coordinate is time and whose y-coordinate is the number of disk pages currently in use. Peek will periodically check the number of allocated disk pages, and move the plotted curve to the left, adding the most recent datum to the right of the graph.

It is possible to force PEEK to immediately sample the number of allocated disk pages. The first such request is made by hitting the "RETURN" key. Subsequent requests are made by lightpenning the plotted curve.

To exit from PEEK, type (CTRL-*). This is the program break key, and is typed by simultaneously depressing the key marked "CTRL" and the key marked "*".

PEEK will attempt to open a directory called "ROOTS". A directory with this name is present in user 2's root directory. It contains an entry for the root directory belonging to each user who is currently logged into the system. PEEK makes use of this to display a list of these users. If an ordinary user has a file or directory with the name "ROOTS" in his root directory, PEEK may display some anamalous information.

XPL

XPL is a language designed for the implementation of language translators and other non-numeric utility programs. The compiler for XPL is itself written in XPL, and the language is also being used, where appropriate, for the implementation of various system processors (such as the I/O Job). The compiler automatically generates re-entrant code.

XPL is a PL/1-derivative language containing only those language constructs thought essential to the implementation of compilers. It contains no floating point data type, and does not allow arrays of more than one dimension. It does contain a string data type, with a mechanism for dynamically allocating storage space for strings, a string concatenation operator, a "substring" function ("SUBSTR"), a "BYTE" function, and parameterless macros. The language includes an "IF-THEN-ELSE" statement, "DO-STEP" statement, "DO-WHILE" statement, and a "DO-CASE" statement. With the exception of strings, storage is allocated statically; recursive procedures are not allowed.

In addition to the compiler, an XPL program is available (the processor "SLRK" - see page 56) which will automatically generate a sy ntax analysis for any of a large class of programming language grammars. A source language library ("XPL-SIGLIB") is also maintained. At the present time it contains chiefly the routine "COMPACTIFY", which d ynamically maintains the storage for strings.

XPL was originally designed for the IBM 360 line of computers by McKeeman, et. al. (16) and the architecture of those machines has had a great effect on the structure of the language. Fortunately, the Sigma-7 is very similar to a 360 and the local implementation of XPL differs very little from the language defined in (16), which we shall call XPL-360. The local implementation of XPL should be able to correctly compile any program written in XPL-360. To be useable, of course, such a program normally requires the addition of a small number of calls to the implicitly defined procedure "ASSIGN" to effect a connection with input and output files or the display console.

As indicated above, the person wishing to make use of the Sigma-7's XPL should obtain a copy of "A Compiler Generator" (16). In addition, a manual describing the idiosyncracies of the local version of XPL (15) is available from John Beatty or Gain Wong.

To run the XPL compiler, type

EXECUTE XPL

You will then be asked to specify a source file, a listing file, and a binary file. The source file is a Gordo text file containing the program to be compiled. A compilation listing will be placed in the listing file; Code which is generated for the program will be written into the binary file. This file is in the "standard executable format", and may be directly executed by means of the Master "EXECUTE" command. a new file will be created for the listing or binary files if one of the indica ted name does not already exist.

The XPL compiler does not run efficiently on the Sigma-7 in its current configuration as XPL's working set is somewhat larger than the physical core currently available for swapping. Long programs running without competition should compile at 600-650 cards per minute. Any competition will substantially reduce the compilation rate (interactive users will of course be run before the compiler - any other computation bound program will repeatedly force the system to swap the XPL compiler out of core). This problem will be greatly alleviated when the additional 96K of core arrives. Users of XPL should also be aware that the compiler requires substantial initialization before it is prepared to begin work; hence very short programs will also seem to compile a bit slowly.

SLRK

"SLRK" is an XPL program written by Franklin DeRemer which determines if a supplied context free grammar is SLR(0) or SLR(1). If this proves to be the case, the program will punch tables which define a parser for the grammar (ie, a deterministic pushdown automaton). Given the BNF for a programming language (such as XPL), it is intended that "SLRK" generate a parser which can be inserted into a compiler for the language in order to automate the process of syntactic analysis. SLRK is an integral part of the XPL compiler generating system described in (16), and used on the Sigma-7 to generate the parser for the XPL compiler. [However, a different parsing method is detailed in (16). It has been replaced with an SLRK parser-generator.] SLR(k) grammars were introduced by DeRemer in (20), and are also discussed in (21) and (22).

To run SLRK, type

```
EXECUTE SLRK
```

The program then asks in turn for the "grammar file", "listing file", and "punch file". The grammar file is a text file containing the grammar to be analysed. The results of the analysis are recorded in the listing file. The punch file will contain a specification of the parser for the language, in the form of XPL variable declarations.

INPUT - THE GRAMMAR FILE

The grammar file consists of control cards and grammar specification cards. The comment cards must have a "down-delta" (keypunch: shift C) in column one. The next character on the control card (column two) designates a control toggle. The toggle specified is complemented. That is, if it was on it is turned off, and vice versa. The control toggles currently in use and their initial settings are as follows:

I	off	Copy the grammar file to the output listing
G	on	Print the grammar neatly in the output file
C	off	Print the configuration sets
F	on	Print the CFSM(CharacteristicFiniteStateMachine)
L	on	Print the lookahead sets
D	on	Print the DPDA(DeterministicPushDownAutomaton)
P	off	Punch the parser tables

In addition, the appearance of "EOG" in columns 2-4 of a control card flags the end of a grammar. The cards following an "EOG" card are analysed separately as a distinct grammar. By this means several grammars can be analysed with a single execution of the program. Do NOT place an "EOG" card after the last grammar in the grammar file. NO EOG card is required if the grammar file contains but a single grammar.

The grammar specification cards describe the grammar to be analysed. Each card describes one production of the grammar. A symbol which starts in column one is regarded as the left-hand side of the production. The right-hand side follows on the same card. All symbols are separated by a blank. The definition operator used in BNF ("::=") or in language theory ("->") must not appear. Alternate productions for the same left hand side must appear on immediately succeeding cards; the left hand side does not appear on such alternatives, and column one is left blank to indicate that this is an alternative right-hand side for the last specified left-hand side.

An atomic symbol of the grammar may be any one of the following items:

- A left corner bracket surrounded on both sides by a blank (thus allowing the use of "<" in the language described).

- Any string of non-blank symbols which does not begin with a left corner bracket ("<") and terminated by a blank or the end of a card.

- A left corner bracket ("<") followed immediately by a non-blank character, extending up to and including the next instance of a right corner bracket (">").

Notice that the use of a corner bracket does not imply that the symbol is a non-terminal. Rather, any symbol which does not occur on the left of some production is regarded as terminal. The first symbol which appears only on the left hand side is regarded as the goal symbol (start symbol). If there is no symbol which appears only on the left, then the left hand side of the first production is taken as the start symbol.

Thus the following grammar, specified in BNF ..

```
<arithmetic expression>  ::=  <arithmetic expression>  +  <term>
<arithmetic expression>  ::=  <termn>
<term>     ::=  <term>  *  <factor>
```

SLRK

```
<term>    ::= <factor>
<factor>  ::= <identifer>
<factor>  ::= (  <arithmetic expression>  )
```

might be input as

```
<arithmetic expression>  <arithmetic expression>  +  <term>
      <term>
<term>  <term>  *  <factor>
        <factor>
<factor>  <identifier>
          (  <arithmetic expression>  )
```

Alternatively, we might express the above grammar according to the conventions of language theory as

```
E  ->  E + T
E  ->  T
T  ->  T * F
T  ->  F
F  ->  I
F  ->  ( E )
```

which would be supplied to SLRK in the form

```
E E + T
 T
T T * F
 F
F I
 ( E )
```

There is no provision at the present time for using more than one card to specify a production.

TAPES

This utility routine is designed to service most of the needs of users desiring backup storage on the Sigma-7's magnetic tape units. More specifically, it provides a convenient means whereby the user may transfer gordo files to and from standard gordo tapes, and append files to an already existing Gordo tape. It can also be used to communicate with the Octopus computers via standard BCD tapes or Gordo tapes. The routine has been designed to handle many error situations automatically. All input is via the alphanumeric keyboard or the console lightpen. Most commands are issued by selecting an item from a command menu with the lightpen. The program is initiated by typing

EXECUTE TAPES

followed by the "RETURN" key.

BASIC COMMANDS

The basic command menu is displayed whenever the program begins execution, or when it is selected from one of the other command menus. A basic command is selected with the lightpen (as are all commands from any other menu); The pound sign ("#") to the immediate left of the menu item is hit with the lightpen by pressing the lightpen to the screen at the location of the pound sign for the desired menu item. The program will respond by either requesting input or by displaying a new menu. Note that each time a menu item is selected, the lightpen targets (pound signs) are moved to a different position. This is to prevent a user from inadvertantly selecting a menu item from a subsequent menu when a command is processed before he can remove his lightpen from the screen.

The basic command menu is primarily concerned with designating the tape and files for which input or output is desired. The name of the tape requested is displayed in the upper right hand corner of the screen. A list of the file names which have been selected is displayed at the top of the screen.

Messages from the routine are displayed in the center of the screen between two horizontal lines. This is the "message area". Each command will be discussed, with examples, on the following pages. The basic commands (as displayed on the screen) are:

```
#   Set tape name.
#   Add file names.
#   Add file names from root directory.
#   Delete file names.
#   Delete all file names.
#   Process tape.
#   HELP! (instructions).
#   Exit to System.
```

SET TAPE NAME.

Upon selection of this command, the message

TYPE NAME OF TAPE.

will appear on the center of the display screen. The user should then input the name of the tape to be used. Any string of ten (or fewer) characters is acceptable. If less than ten characters are typed, the name will be left-justified and padded on the right with blanks. Any additional characters will be ignored. The routine will then return to the basic menu for a new command.

ADD FILE NAMES.

Upon selection of this command, the message

TYPE A FILE NAME.

will appear in the message area. The user should then input the name of a file for which he desires tape input or output. Any string of ten (or fewer) characters may be supplied at this point. The routine will continue to append file names to the list until either the maximum allowable number of file names have been added (currently 50) or until a null input line (return) is supplied. The routine will then return to the basic menu for a new command. Note that no action is taken at this point with respect to the files named in the list; No check is made to verify that the file does, indeed, exist.

ADD FILE NAMES FROM ROOT DIRECTORY.

Upon selection of this command, every file in the user's root directory (except "PUBLIC" and "USERCOMPF") is added to the file list. The routine will then return to the basic menu for a new command. Note that the files "**RFLWS***" and "DECODE", which are created by the tapes routine, will also appear in the file list as a result of selecting this command, and should normally be deleted from the list be means of the "DELETE FILE NAME" command.

DELETE FILE NAMES.

Upon selection of this command the message

LIGHTPEN FILE NAMES TO DELETE.

will appear in the message area at the center of the display screen. Lightpenning a file name which is listed at the top of the display screen will cause it to be deleted from the list. The message

DELETE NO FILE

will appear in the bottom half of the screen. Lightpenning the pound sign ("#") at its left will cause the routine to exit from the delete mode and return to the basic command menu.

DELETE ALL FILE NAMES.

Upon selection of this command, every file currently in the file list is deleted and the routine returns to the basic command menu to await the selection of another command.

PROCESS TAPE.

Selection of this command causes a new command menu (the "processing menu", described below) to be displayed on the screen. It is by means of the processing menu that the user requests that a tape be read, written, or have its contents listed.

HELP (INSTRUCTIONS).

Selection of this command causes the routine to enter the help mode. While in the help mode, the message "HELP" is flashed vertically at the left edge of the screen. (this message will remain on the screen until the program exits from the help mode.) Instructions as to how the help package is used are displayed on the screen. When they have been read, they should be lightpenned. They will then be replaced by the basic command menu. At this point, lightpenning any command will result in the display of a description of the use of that command. Lightpenning any such instructions will cause a return to the basic command menu. While in the help mode, the "PROCESS TAPE" command may be selected; this results in the display of the processing menu, for which help instructions are similarly available.

To exit from the help mode, return to the basic command menu and select the help command with the lightpen.

TAPE PROCESSING COMMANDS

The processing menu is displayed upon selection of the basic command "PROCESS TAPE". As a result of selecting any of the commands on this menu, the appropriate tape I/O request is made of the tape handler (which is one of the processes in the I/O Job). The message

I/O has been requested.

will appear in the message area at the center of the screen and will remain there until the requested I/o has been completed.

If an error is encountered, an appropriate diagnostic will appear in the message area. It will remain until it is lightpenned, at which point the routine will return in the normal fashion to the basic command menu (or another error message will appear).

The following processing commands are currently available:

- Read files from tape.
- Read text files from BCD tape.
- Write files onto new tape.
- Write text files onto new BCD tape.
- Append files to old tape.
- Examine contents of tape.
- Return to file menu.

READ FILES FROM TAPE.

The tape is assumed to be a standard gordo tape. The files requested are read from the tape and placed in the user's root directory. All files will be read at the current security level and will have a lifetime of three hours.

READ TEXT FILES FROM BCD TAPE.

The files in the file list will be read from a BCD tape. A BCD tape is a sequence of files. Each file is a sequence of 120 character BCD records. The tape is assumed to have been written in standard laboratory 6-bit ASCII. Files are separated by end-of-file marks. Since the files are not labeled, they are associated with names in the file list in the order given; the first file read is given the name which is first in the file list. The file created will be a standard Gordo text file.

WRITE FILES ONTO NEW TAPE.

The files requested are written onto a new standard Gordo tape. Any data previously on the tape is overwritten.

WRITE TEXT FILES ONTO NEW BCD TAPE.

The files specified are written onto a new tape in standard BCD format. Only text files are kosher. The requested files are written in the order requested.

APPEND FILES TO OLD TAPE.

The tape must already have been formated as a standard Gordo tape. The requested files are added to it (by a previous "write files" operation). If a file by the same name already exists on the tape, the pre-existing copy of that file is lost. If there is room, the new copy is written on top of the old; otherwise the new copy is written after the last file already on the tape.

EXAMINE CONTENTS OF TAPE.

The tape is loaded and its directory read. The file names entered in the directory are added to the file list displayed at the top of the screen.

RETURN TO FILE MENU.

Selection of this command will cause the basic command menu to be displayed in place of the processing menu, so that the file list may be corrected if an error is discovered after selecting the processing menu.

ERROR MESSAGES

Error messages are displayed in the center of the console screen. They will remain until lightpened.

COPYING TAPES

The normal procedure for copying a tape is to

1. Examine the contents of the tape. The result of this will be to add the name of every file on the tape to the file list.
2. Read the files now named in the file list into the system.
3. Write the files, which are now entered in your root directory, onto a new tape.
4. Delete the files from your root directory (to conserve disk space).

At the present time there is no convenient way to transfer files one by one from an old tape to a new one. An anticipated change in the system will eliminate this difficulty, but not in the immediate future. As a result, all of the files to be transferred must be read before any are written onto the new tape. Caution should be used to avoid filling the disk.

THE PERSPICACIOUS USE OF TAPES

It is not safe to make a habit of appending files to old tapes for two reasons. Firstly, there is always a danger of unintentionally writing on top of a tape instead of appending to it. Secondly, if a tape should be damaged, or become unreadable for one reason or another, it is desireable to have backup copies of the files it contains which are not excessively antiquated. It is therefore advisable to write a significantly modified file onto a new tape. To be completely safe, a valuable file should be read back and checked (for example, by the editor). It is easiest to keep track of these tapes if they have a common name together with a sequence number, such as "6-ASYS25". Alternatively, some number of tapes ("n") may be used in round robin fashion so that the last "n" versions of a file are preserved.

TAPEHAND

GRAPHICS

FILE FORMAT CONVENTIONS

SYSTEM CONVENTIONS

In the context of Gordo, a "file" may refer to either a directory or a data file; the term "data file" is used to refer to either program data in the conventional sense, or program code (which, after all, is data to a compiler). Files are maintained on the swapping disk, which is allocated in blocks of 512 words each called "disk pages." (Notice that a memory page and a disk page are the same size.)

Directory files consist of only one disk page, in which there is room to specify up to 100 directory entries.

The first disk page of a data file (disk page 0) is always the file header. It contains various system information, including a map which indicates which file pages have been allocated disk space, and if allocated, where they may be found on disk. This disk map has room for only 1000 entries. The i-th entry contains the disk address of page i of the file, or zero if page i has not been allocated disk space. Entry 0 points to the file header page (itself). Hence a data file may consist of at most 999 pages of data, numbered 1 through 999.

A- and B-system do not enforce any further restrictions on the structure of files. Certain of the processors, such as the editor and Master, do expect the contents of a file to obey certain conventions. We shall now describe the four file types of which system processors make use.

TEXT FILES ('E0E0E0E0')

All text is represented in the 8-bit Gordo-ASCII character code (see table 1). Disk page 1 of the text file contains a text file header. A text page map contained in this header is used to order the subsequent pages of text within the file. The n-th map entry contains the page number of the file page associated with the n-th page of text logically contained in the file. A zero map entry denotes an unused page.

Word 0 of the header page for a text file contains the hexadecimal value 'E0E0E0E0' which identifies the file as a text file. Word 1 contains the number of the text map entry which contains the last line of text in the file. (The first map entry is 1, and the last possible entry is 998, since of the 1000 disk pages which may be contained in the largest possible file, one is used by the system for the system file header and one is used by convention for the text file header.) Word 2

contains a count (possibly zero) of the number of lines of text in the file. The text map begins in the sixth word, and continues for 508 entries. The text map is used primarily by the text editor and for most purposes it can be set up with the first entry having the value of two (ie, the first page of text being in page 2 of the file) and increasing by one for consecutive entries.

All text pages have a similar format. A text page consists of a sequence of text lines (described below) followed by an end-of-page indication. The only exception is the last page of text (pointed to by the text header as described above), which is terminated with an end-of-file indicator. A page may contain zero lines. All lines are completely contained within a page so that no line overlaps a page boundary.

A text line begins with a byte containing a count of the number of bytes needed to represent the line of text. The string of bytes forming this representation follows the count byte, and is terminated by a carriage return. The carriage return is included in the count. The count byte is not included in the count. A count of 0 represents padding between valid lines; the existence or non-existence of zeros between lines does not effect the logical contents of the file. A blank line has a count of one and is terminated by a carriage return. The count must be less than 254. A count of 254 or 255 will be considered an end-of-file or end-of-page indication, respectively.

Within a line of text any 8-bit character may be used with four exceptions: carriage return, horizontal tab, vertical tab, and the escape character.

The escape character is used to enter one of these special characters within a line. When an escape character is encountered within a line the byte immediately following is treated as text and is not considered a special character. Because a carriage return denotes end-of-line, it must be escaped in order to appear within a line as text.

The horizontal tab character is skipped and the byte immediately following is treated as a non-negative count of the number of blanks to be inserted in the line before the next character. Note that zero is a legitimate count.

The vertical tab character is used primarily for printer control information. It is skipped and the byte immediately following is treated as a format control code according to the following conventions:

HEX VALUE	FUNCTION
'60','e0'	No line feed
'C1'	Skip one line
.	.
.	.
.	.
'Cf'	Skip fifteen lines
'F0'	Top of page
'F1'	Bottom of page
'F2'	Skip to channel 2
.	.
.	.
.	.
'F7'	Skip to channel 7

Most programs ignore the presence of vertical tab characters. The following character codes have special meanings:

HEX VALUE	NAME	FUNCTION
'A8'	Escape	To ignore the special meaning of special characters.
'8F'	Carriage return	End-of-line
'FC'	Vertical tab	Printer format control
'FD'	Horizontal tab	Insert blanks into line
'FE'	EOF	End-Of-File
'FF'	EOP	End-Of-Page

The characters which must be typed in order to enter the above from a console keyboard may be found in Appendix 4.

BINARY FILES (D0D0D0D0')

A binary file contains binary card images. All card images are 80 columns (960 bits) in length. Each page of the file contains from zero to seventeen column-binary card images. The last page of card images contains an end-of-file flag.

Binary files are produced by SYMBOL, FORTRAN, and DLOADX. They may be punched by means of the Master "PUNCH" command.

Word 0 of each page contains the hexadecimal value 'C0C0C0D0'. This identifies the page as one containing column-binary images. Word 1 contains a count (0-17) of the number of card images in the page. If the high-order bit of the count is a one then the current page is the last page of card images. The remaining 510 words of the page contain the column-binary images in consecutive words.

OLD EXECUTABLE FILES ('F0F0F0F0')

An executable file contains all of the information necessary to load a progra m. Old executable files are created from (DLOADed) binary files by FLD a nd are labeled with the id 'F0F0F0F0'. They cannot be punched.

The second page of the file (page 1) contains an executable file header which tells Master how to couple the pages of the file into virtual space in order to run the program. The words in the header are used as follows:

word	name	function
0	id	labels file as (old) executable
1	file	# of first disk page to be loaded
2	virtual	# of first virtual page to be loaded
3	number	total # of pages to be loaded
4	start	entry point

When an (old) executable file is loaded the number of pages specified will be coupled consecutively into virtual space beginning with the file and virtual pages indicated. The remaining pages of the file contain the program to be executed. This scheme lacks flexibility; hence the standard executable file format (see below) is preferred.

STANDARD EXECUTABLE FILES ('A0A0A0A0')

Files of this format are created by the XPL compiler. The second page of the file (page 1) contains an executable file header which tells Master how to couple the pages of the file into virtual space in order to run the program.

Word 0 contains the hexadecimal value '0a0a0a0a', which identifies the file as a standard executable file. Word 1 contains the entry point of the program. Words '100' through '1ff' are virtual page frame control words. They control the coupling of a page into virtual address space according to the following conventions:

BITS	USE
0-7	This specifies the type of load to be performed for the corresponding virtual page frame: = 0, do not load the corresponding virtual page frame. = 1, couple a file page from the executable file into the corresponding virtual page frame. = 2, couple a scratch page into the corresponding virtual page frame. = 3, couple a scratch page into the corresponding virtual page frame and copy the contents of the indicated executable file page into the scratch page.
8-15	The access level at which the virtual page frame is to be coupled.
16-31	The page number of the executable file page which is to be coupled or copied into a scratch page. The scratch page used is always the virtual page frame number + 1. The scratch file resides in keyword 10 and is maintained by Master.

TAPE FORMAT CONVENTIONS

GORDO TAPES

INTRODUCTION

This section describes the format and usage of standard Gordo tapes. Standard Gordo tapes act as an extension of the Gordo file structure, and have a format peculiar to the Gordo system. Because tapes written in this standard format are converted directly by the i/o handlers into Gordo files, the user does not have to perform messy conversion operations after reading these tapes. Whenever practical, such operations should be performed on the machine originating the tapes instead of the Sigma-7, both because the Gordo system is meant to function interactively and does not provide fast service for lengthy conversions, and because tapes are typically written once but read often.

PHYSICAL CHARACTERISTICS

All Gordo tapes are 7-track binary tapes and are written in odd parity. All records are assumed to be 512 words long. (Recall that a Sigma-7 word is 32 bits in length.) For all i/o operations additional bits contained in a record will be ignored. The presence of such extraneous bits will not be considered an error and no indication of their presence will be given. This enables the Gordo system to read tapes written by machines with different word sizes. Gordo tapes may have a density of either 200, 556, or 800 BPI. The density should be clearly marked on the tape reel at the time it is written.

STANDARD TAPE FORMATS

A standard Gordo tape is a Gordo tape with a series of records and end-of-file marks (EOF's) which follow the standard tape format given below. Each tape record contains 16,384 bits, enough to fill exactly one Sigma-7 page. (If a tape record is longer that one page, the extra information will be ignored by the i/o handler.)

There are three types of fields allowed on standard Gordo tapes: leader, directories, and files. Each is described in a following section. The overall tape format is as follows:

Tape Formats

```
# # # (leader)

                # # # (directory) # # (file 1)

                # # # (directory) # # (file 2)

                .
                .
                .

                # # # (directory) # # (file n)

# # # (directory) # # # #
```

where "#" represents an end-of-file mark. Note that the minimum tape consists of three EOF's, a leader, three EOF's, a directory, and then four EOF's. For each file on the tape, a new copy of the directory, followed by two EOF's, the file, and then three EOF's is written on the tape before the final directory. Thus there is always one more directory on the tape than there are files on the tape.

LEADER

The leader consists of two records. The first 80 bits of the first record is the name of the tape, written in the Gordo 8-bit ASCII code used for all text in the Gordo system. All remaining bits within both records are ignored.

DIRECTORY

A directory is one record long and has the following format (see Figure 1) - the notation "n.m" denotes byte m of word n; all sizes are in bytes:

loc	name	use
0.0(4)	CREATOR	The job number of the tape's creator.

1.0(2)	FREE	The word offset (relative to the first word of the directory) of the first free entry in the directory
2.0(2)	FIRST	The word offset (relative to the first word of the directory) of the first directory entry which is in use.
3.0(2)	PEOF	The number of EOF's preceeding this directory on th e tape.
3.2(2)	TEOF	The total number of EOF's on the tape.

Beginning in word 4 of the directory is a table of 100 directory entries, each of which occupies 5 words. Each entry has the following format:

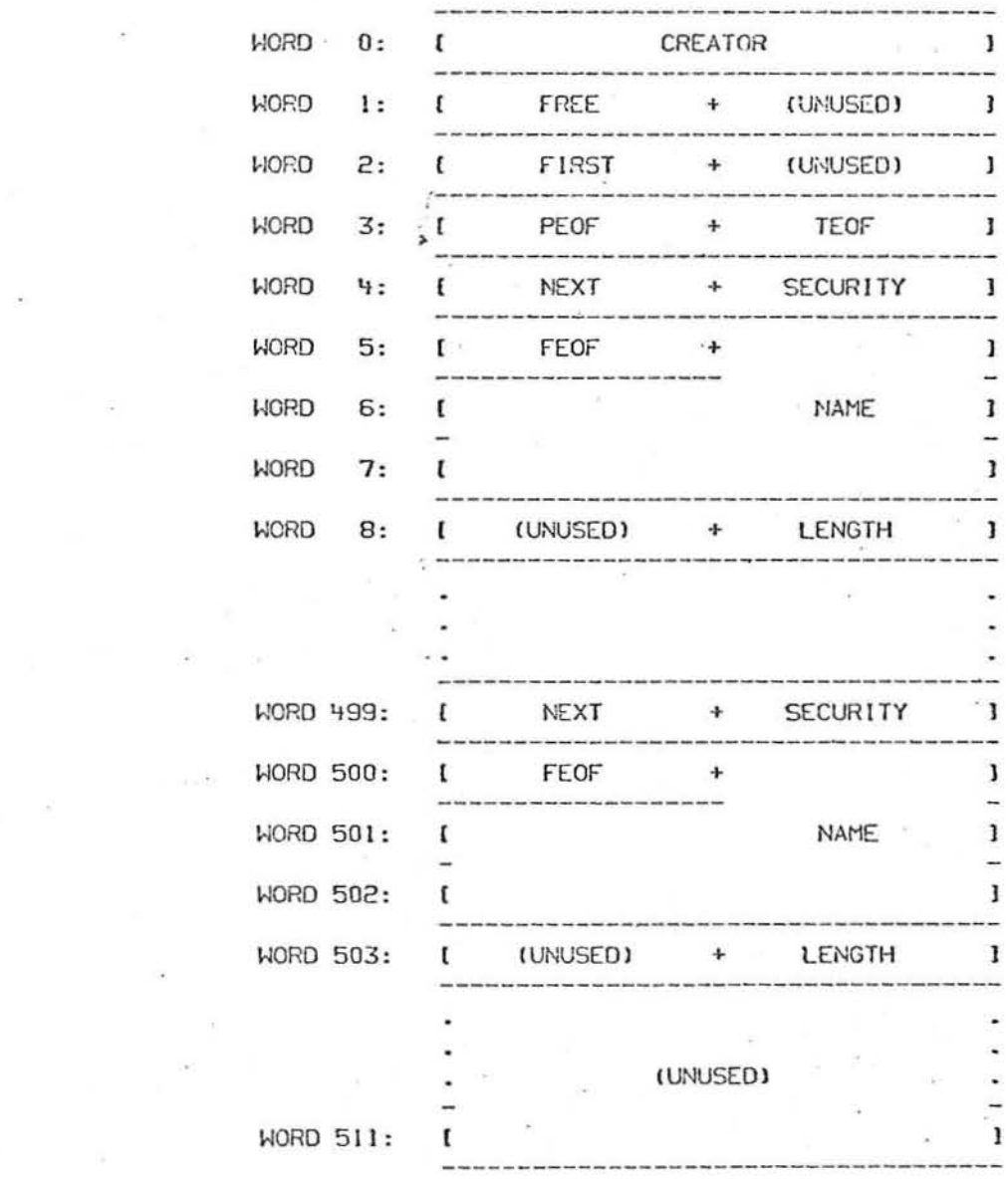
loc	name	use
0.0(2)	NEXT	word index in the directory of the next entry
0.2(2)	SECURITY	Security level of the file.
1.0(2)	FEOF	The number of EOF's preceeding the file.
1.2(10)	NAME	A 10 character (80 bit) file name, in the Gordo-Ascii character code.
4.2(2)	LENGTH	The number of records in the file.

The use of directories facilitates placing more than one file on a tape. The directory is repeated before each new file to provide error checking redundancy.

FILES

A file may have any number of records from 1 to 1000. The first record of a tape file is always the tape file header and determines how the remaining records are to be mapped into the file by the I/o handler. The first five words of the file header are unused. Beginning in the rightmost halfword of the sixth word of the record is a table of 999 halfword entries. Page "i" of a file is recorded on the tape if and only if the entry "I" of the file header is non-zero. The records representing a file are read

consecutively; each record is associated with the next non-zero table entry. Thus if a file header has non-zero entries for pages 13, 200, and 998 then the tape file will have four records - a file header and three data records. The second record will be associated with page 13, the third with page 200, and the fourth with page 998.



Note that if a file page is to be all zero, the most efficient use of the Gordo system will be to leave the entry in the file header for that page zero, and not place the page on the tape. This not only saves tape and i/o time, but when the tape is read in, no page will be assigned from the disk until the all-zero page is actually referenced by a program. Only at that time will an all-zero page be created. Thus disk space is conserved until actually needed.

More importantly, if all files are arbitrarily created on tape with the maximum of 1000 pages, at most one can be read at a time since the total number of disk pages available to all users on the Gordo system is 2624. This suggests that whereas 1000 pages are allowed, this is a totally impractical number and most files should be considerably smaller.

SPECIAL FILES

Sufficient information has been provided at this point to enable a user unfamiliar with the Gordo system to create a tape on a machine other than the Sigma-7 which can be read by the Gordo system, and which will result in the creation of arbitrarily structured Gordo disk files. Presumably the average user is more interested in certain common types of files, such as text files, binary card image files, and executable files. The structure of such files is entirely determined by the processors which make use of them. They are described in a separate section of this manual (see "DISK FILE FORMATS").

ALIEN TAPES

WRITING ALIEN TAPES

One writes an "alien tape" by placing a directory in the mailbox "PUBLIC.ALIENOUT", which has an entry for a file having a particular format. Namely, it must consist of an arbitrary number of records, packed sequentially in the file, each of which is preceeded by a word (beginning on a word boundary) whose contents i s the number of bytes in the following record. A tape record will be written for each record so specified in the file; the records will be written in the order of their occurence within the file. The word containing the byte count is not included in the record which is written.

READING ALIEN TAPES

The structure of the disk file created for the tape will be the same as that described above for writing alien tapes.

SYSTEM OPERATIONS

INTRODUCTION

The Sigma-7 is provided with operator support only during normal working hours ("day shift"). The machine is generally available during off hours, provided that preventative maintenance or system debugging is not under way. In order to use the machine at such times, the user must know how to perform some of the functions normally taken care of by operators. This section has therefore been written with several goals in mind. It is hoped that the reader will be able to load and operate a system himself, if that is necessary, and that he will benefit from a greater understanding of a few of the nuts and bolts of system operation.

The reader should understand that users are NOT allowed in the operations area during day shift, as this invariably interferes with the operators.

MOUNTING TAPES

If you do not know how to thread a tape through the read/write heads, have someone show you how to do it. It is fairly simply, but is a bit difficult to describe here. ALWAYS CHECK TO SEE THAT THE TAPE UNIT IS SET TO THE CORRECT DENSITY. There are now three buttons to push, in the following order:

"LOAD" - this advances the tape so that the load point is under the read/write heads.

"ATTENTION - this informs the tape processors of the unit on which you have just hung the tape.

"START" - this brings the unit on-line, so that it may be read or written by the tape processor.

ABORTING PERIPHERAL DEVICES

SKELETON TAPE:

A skeleton tape contains the minimum amount of information necessary for a running system. It is divided into three sections. The first is a bootstrap loader (one record). The second is a 18K record which contains the A-system. The third records the initial contents of the swapping disk. This section is composed of a sequence of 513 word records; Each of these represents the contents of a disk page which must be present on disk in order for the skeleton system to run. Only non-zero pages are present. The first word of each record contains the page number of the disk page associated with the record. The last record is a dummy; it indicates a page number of -1, flagging the end of this section section of the tape (called initial-disk).

The bootstrap loader reads the A-system into core; The A-system initializer is then responsible for transfering initial-disk from tape to disk, by means of the "T" or "S" commands (see page 85).

CHECKPOINT TAPE

A checkpoint tape is identical in format to a skeleton tape. However, the initial-disk section contains everything needed for a complete system. A checkpoint tape, as the name implies, is the image of a running system.

PROCESSORS TAPE

A processors tape contains all of the processors needed to make a complete system (eg, the System Loader, I/O Job, Fortran compiler, XPL compiler, Symbol assembler, Master, Edit, etc.). A complete system is made from a skeleton system by reading the contents of the Processors tape (which is in standard Gordo format) and placing the files on this tape into public. This function is performed by the System Loader, which is stored on the Processors tape; the System Bootstrap Loader (SBL), which is part of the initial-disk section of a skeleton system, is responsible for reading the System Loader and I/O Job off of the Processors tape, and entering them into PUBLIC and/or Super-user's root directory, as is appropriate.

LOADING A SYSTEM

- Mount the system tape (which is a checkpoint tape) on unit 0, hit "LOAD " and "START".

- Set sense switches 1, 2, and 4 down, and Set sense switch 3 up. This cuts off the Sigma-3. (The sense switches are numbered from left to right.)

- Dial '0e0' on the unit address Dial

- Move the "RUN/IDLE/STEP" ("RIS") switch to idle

- Simultaneously depress the "SYS RESET CLEAR" and "CPU RESET CLEAR" buttons for a second or more (zeroes core)

- Depress the "I/O RESET" button

- Depress the "LOAD" button

- Move the RIS switch to run This will read A-system from the system tape.

- When the tape has rewound, type "S". This will read the contents of initial-disk from the system tape.

- When the tape has again rewound, the system should be running and available for general use.

GENERATING A SYSTEM

The following steps enable one to generate a complete system from cards and a Processors tape. Skeleton and checkpoint tapes are created in the process. (We describe a modified version of the sysgen procedure which is currently being implemented.)

1) Place the "3-Card-Loader" (3CL), followed by a binary deck for A-system, in the card reader. Then perform a hardware boostrap load from the card reader by performing the following operations on the Sigma-7 console:

   - dial '003' on the unit address dial

   - move the "RUN/IDLE/STEP" ("RIS") switch to idle

   - turn the card reader on and hit "START"

   - simultaneously depress the "SYS RESET CLEAR" and "CPU RESET CLEAR" buttons for a second or more (zeroes core)

- depress the "I/O RESET" button

- depress the "LOAD" button

- move the RIS switch to run

The effect of the above is to read one record from the device selected on the unit dial into a fixed core location. In this case, the first card of the 3CL is read from the card reader. The code contained on this card then causes the second and third cards of the 3CL to be read from the card reader.

2) The 3CL then reads a binary deck of the A-system into core from the card reader. Note that each card of a binary deck indicates the address at which its contents are to be loaded. Two points of interest:

   - The initial, assembled A-system must contain

     . a JOB TABLE for the System Bootstrap Loader (SBL)
     . a PACT for the System Bootstrap Loader (SBL)
     . an entry for the SBL's PACT on the short quantum queue

   - The A-system is, of course, locked down in core. It is represented by a code file, for which there is a standard file header (allowing read-only access), but A-system code pages are assigned to non-existent disk pages to avoid tying up disk space.

3) Control is then passed to the A-system initializer, which is used to read the contents of initial disk from either a skeleton tape and/or the card reader. Initial disk contains

   - The directory ROOTS, which at any given time will contain an entry for the root directory of each user currently (or recently) active in the system.

   - The B-system code file

   - The SBL code file

   - The computation file for job 1, which contains SBL in process 1. The computation file contains three pages:

. a file header
. a sub-pact for process 1 (S8L)
. a process descriptor page (PCP) for the job

Initial disk may be read from a skeleton tape by means of a "T" command, or from the card reader by means of an "L" command. Alterations may conveniently be made by reading an old initial disk from a skeleton tape, and then loading modified pages from the card reader, thus overlaying the disk pages to be replaced.

ROOTS is included on initial-disk so that the save-restore routines, which recover the contents of the disk after a non-recoverable system crash, may know where to find each user's root directory.

4) At this point a skeleton tape can be made on unit 0 by means of the "H" and "U" A-system commands (see page 85).

5) Using the "G" command (see page 85), the system is started. Since the PACT for S8L is assembled on the short quantum queue, it begins to run. S8L is a machine language program which initially contains open keywords for the A-system code file, B-system code file, its own computation file, and its own code file. It is responsible for:

- Creating a code file for the System Loader (SL) and for the I/O Job.

- Reading the System Loader and I/O Job from the Processors Tape, which is in standard Gordo format.

- Loading the System Loader into its own virtual address space.

6) The SL, which now has control, is an XPL program. It is responsible for:

- Creating the initial file structure

- Reading the system processors, which are on the "PROCESSORS" tape and placing them, as appropriate, in user-2's root directory (user-2 is a system user) and/or in PUBLIC.

- Starting the garbage collector and the Logger.

7) The system is now complete, and may be check-pointed on tape by raising sense switch 1 and using the A-system "H" and "U" commands (see pages 85 and 56).

In order to generate a system from a skeleton tape, step 1) above is replaced by

1) Mount the skeleton tape on unit 0, hit "LOAD" and "START". Then perform a hardware boostrap load from the card reader by performing the following operations on the Sigma-7 console:

   - dial '0E0' on the unit address dial
   - move the "RUN/IDLE/STEP" ("RIS") switch to idle
   - simultaneously depress the "SYSTEM RESET CLEAR" and "CPU RESET CLEAR" buttons for a second or more (zeroes core)
   - depress the "I/O RESET CLEAR" button
   - depress the "LOAD" button
   - move the RIS switch to run

   The effect of the above is to read one record from the device selected on the unit dial into a fixed core location. In this case, the first record is a small bootstrap loader which reads the next record from the tape into core at word '200'. This record (the second on the skeleton tape), is an image of A-system. Control is then transfered to the A-system initializer.

Steps 2 - 7 are unchanged.

The following is a map of initial disk:

page # contents

0-F ... B-system code

10 ... B-system code file header page

11 ... A-system code file header page

12 ... Job 1 computation file header page

13 ... Job 1 processor descriptor page (PDP)

14 ... Job 1 Process 1 Sub-pact (for SBL)

15 ... System Bootstrap Loader code file header page

16-19 ... System Bootstrap Loader code (released after sysgen)

CHECKPOINTING A SYSTEM

THE A-SYSTEM INITIALIZER

BIBLIOGRAPHY

(1) F. N. Fritsch and R. F. Hausman, "On the Documentation of Computer Programs", Lawrence Livermore Laboratory, UCID Report 30043, March 1972

(2) R. W. Watson, "Timesharing System Design Concepts", McGraw Hill, 1970

(3) Richard E. Hamming, "One man's view of computer science", JACM 1969 vol. 16, # 1, p. 10 (The Turing Lecture for 1969)

(4) D. E. Knuth, "An Empirical Study of Fortran Programs," SOFTWARE - PRACTICE AND EXPERIENCE, vol 1, 1971, pp 105-133.

(5) W. G. Alexander, "How a Programming Language is Used", Computer Systems Research Group Technical Report CSRG -10 , 1972, University of Toronto.

(6) E. Satterwaite, "Debugging Tools for High Level Languages", SOFTWARE - PRACTICE AND EXPERIENCE, vol 2, number 3, July-September 1972.

(7) Organick, Elliott I "The Multics System" - The MIT Press - Cambridge, Massachusetts - 1972

(8) Anderson gb,Bertran KR, Conn RW, Malmquist KO, Millstein RE, Tokubo S - "DESIGN FOR A TIME-SHARED GRAPHICS FACILITY" - UCRL 70969 - FEBRUARY 1968

(9) CONN RW "AN OPERATING SYSTEM FOR THE SIGMA-7 COMPUTER" - CIE Meeting 47 - 14 August 1968

(10) Anderson, Gary - "Programming for a Small Paged Environment" - CIE Meeting 53 - 6 November 1968

(11) Anderson, Gary - "A Brief Discussion of Gordo" - UCRL 71365 - September 1968

(12) Bertran, KR - "A Time-shared Environment for the Study and Application of Man-Machine Interaction Through Graphics" - UCRL 71722 - May 1969

(13) Malmquist KO - "GORDO" - UCRL 71788 - May 1969

(14) "SIGMA 7 - SIGMA 3 FORTRAN IV LIBRARY USER'S MANUAL"

(15) Wetherell, Charles - "XPL ON THE SIGMA 7"

(16) McKeeman w, Horning j, and Wortman d - "A Compiler generator" - Prentice-Hall, Inc. - 1970

(17) "XDS SIGMA-7 COMPUTER REFERENCE MANUAL"

(18) "XDS SIGMA 5/7 SYMBOL/META-SYMBOL REFERENCE MANUAL"

(19) "XDS SIGMA 5/7 FORTRAN IV-H REFERENCE MANUAL"

(20) DeRemer, Franklin Lewis - "Practical Translators for LR(k) Languages - MAC TR-65 (PhD thesis) - Project MAC, Massachusetts Institute of Technology - 24 October 1979

(21) DeRemer, Franklin Lewis - "Simple LR(k) Grammars" - CACM July 1971, Vol 14, No 7

(22) Anderson T, Eve J, and Horning J J - "Efficient LR(1) Parsers - University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series 24

APPENDIX 1

GLOSSARY

ACCESS

This is a specification of the uses to which the data in a file page may be put. The levels of access are read/write/execute, read/execute, read-only, and no access, coded 0, 1, 2, and 3, respectively. Read/write/execute is the maximum access level and no access is the minimum (notice that this reverses the numerical order of the coding). A file page available at one access level is automatically available at all lesser access levels. An access level becomes associated with a file page through B-system OPEN and COUPLE calls. Each directory entry for a file has an access level in it and the file may not be opened from that entry at a greater access. Once opened, the file pages are of course referenced only after they have been coupled. But a B-system COUPLE call also contains an access level request and the final access of the coupled page is the minimum of the access level requested in the call and the access at which the file was opened. Any attempt to access a file page at greater than its access level causes a trap. Access levels may be used to control the writing of shared files. In particular, most shared processors, such as FORTRANL XPL, and the text editor, are available to the user at read/execute access only and any attempt to write these files will be rebuffed with a trap. Notice that the access level in a directory entry is a property of that entry and not of the file, so that the same file may have different access levels in different entries. In the same way, a file page may have different access levels in different page frames because of different access levels in coupling requests.

ACCESS LEVEL

(see ACCESS)

ACCESS PATH

This is a general term for a chain of access from one item to another. Under Gordo, the most important access path is that by which a process gains access to a particular word within a file. Each step within the path may be guarded by two access conditions, a security level, and an access level. The typical path is described below. The initial conditions are a directory open in a keyword and the target file entered in the directory.

The first step is to open the file from the directory into a keyword. The file entry must be in the directory at a security level lower than or equal to the current process' security level and the

access level specified in the open call must be less than or equal to that of the entry in the directory. Further, the directory must be open at read access or greater in its keyword. Thus, there are three access and security conditions to be met in this step.

The second step is to couple the needed file page into a virtual page frame. At this step, the page may not be coupled with an access level greater than that at which the file was opened.

Finally, the word within the file page is referenced by some virtual machine instruction. At this step, the access requested by the instruction must be less than or equal to the access level specified by the couple request and by the OPEN call.

In this whole access path, there are five access and security conditions which must be met during transversal of the access path. Note also that the access path is a descriptive idea used to specify the response of the system to various file manipulations and is not implemented directly within the system. Note too, that this whole path need not be traversed for every reference to a file word, but that the effects of steps within the path are cumulative.

Suppose that FILE0 has been entered in DIRECTORY1; that DIRECTORY1 has itself been entered into DIRECTORY2; and that DIRECTORY2 is open in keyword 3. In order to access the contents of FILE0, DIRECTORY2 must be opened from DIRECTORY1 (now open in KEYWORD 3) INTO SOME KEYWORD (possibly keyword 3, in which case DIRECTORY1 is no longer accessible). FILE0 must then be opened into some keyword from the keyword in which DIRECTORY1 is open. At this point FILE0 is accessible, and may be coupled into the user's address space. Of course, all access restrictions must have been satisfied at each open in order for the above to be successful.

ACTUAL COMPUTER

This is the XDS Sigma-7 upon which Gordo is implemented. Unlike many systems, Gordo users have little direct access to the actual computer, but run instead on a virtual computer. Thus, the properties of the actual computer are of little interest to most users. There are, however, three salient aspects of the actual computer of which users should be aware. These are the amount of actual memory available for swapping user programs during timesharing (currently 44 pages), the amount of actual memory available for graphics buffers for users (currently 8 pages), and the amount of disk space available for file storage (currently 2624 pages, of which 500 are always occupied by system code, such as the B-system, XPL compiler, the FORTRAN compiler and library, the SYMBOL assembler, etc.). All of these facilities are likely to grow with the acquisition of new hardware.

ACTUAL MEMORY

This is the hardware core memory attached to the Sigma-7. It is sometimes also referred to as "real" or "physical" memory. There are 32K words of actual memory on the Sigma-7 (soon to be expanded to 128K, the maximum possible). Of this 32K, 10K is pre-empted for A-system use. The remaining 22K may be used to run user processes. The virtual memory of a process is mapped by the system into this actual memory (see paging). This mapping is dynamic and changes as the load on the system changes.

ASLEEP

This is one of the three scheduling states of a process. A sleeping process has suspended computation for the time being. Such a process makes no demands on the resources of the system until it reawakens. A process may put itself to sleep for an indefinite period of time with the B-system SLEEP call and for a fixed period of time with the B-system GIVEUP call. Such a sleeping process may be reawakened by a B-system WAKE call directed towards it by another process, by a console service request, or by the expiration of the sleep time in the case of the B-system GIVEUP call. A process awaiting console input is usually put to sleep, since this avoids strain on the system and the process is sure to be awakened by the input when it comes.

A-SYSTEM

This is the resident portion of the Gordo time-sharing system. A-system provides the basic executive services of process scheduling, disk allocation and control, interrupt service, and graphics service. In addition, it maintains the fiction of the virtual computer for the user. A-system is not directly accessible to the user, but it may be communicated with via B-system.

AWAKE

This is one of the three scheduling states of a process. An awake process is currently in the midst of a computation. The virtual computer of such a process may be likened to an actual computer in its run mode. An awake process may be put to sleep by a B-system SLEEP or

GIVEUP call. It will be made hyperawake by a console service request. The awake state is the normal one for a process.

B-SYSTEM

This is the non-resident (ie, paged), user-oriented portion of the Gordo time-sharing system. B-system maintains the basic information needed to run each process and implements the system calls through which user processes request services of Gordo. B-system exists in a special data file, which may not be manipulated by any ordinary user. pages from that file are coupled into each user's virtual memory from '1E000' to '1FFFF' and are available for execution as normal program material. The code comprising B-system is paged in the normal fashion and is resident in actual memory only when in use by some process. The one B-system file and hence all B-system code, is shared by all user processes. B-system may only be entered through the B-system transfer vector. Also, all of the virtual memory between '1E000' and 1FFFFF is reserved for B-system use and may not be used directly by a user process.

B-SYSTEM CALL

This is a user request for service by the B-system. B-system provides the user interface to the Gordo system through a set of approximately 60 system calls. These calls provide specific monitor services such as the creation of files; opening, deleting, and entering files in directories; coupling of file pages; reading of the keyboard, function box, and light pen; forking of user processes; and so on. Some calls may only be made by processes fulfilling specific conditions (eg, by super-user or a process in the I/O Job). A call is made by an attempt to execute the appropriate location in the B-system transfer vector, usually by means of a branch and link instruction (BAL,14) in the user's program. Arguments for the call are provided by the calling process in virtual computer register 12 and 13 and values are returned in the same registers. One value is usually a success or failure indicator. Most calls also have some side effect, particularly on the console screen or in the keywords or other portions of the sub-pact. B-system always returns to the address in register 15 at the time of the call. B-system calls made by processes not fitting the particular restrictions of the call always result in a return value of 'FF' in register 14.

B-SYSTEM SPACE

This is the portion of virtual memory between 1E000 and '1FFFF'. B-system space is reserved exclusively for the use of B-system and may not be used directly by a user process except for a transfer into the B-system transfer vector or a read reference to that space. In particular, a process may not couple a file page into B-system space, nor may it write into any portion of B-system space. B-system is, of course, free to do what it wills with B-system space.

B-SYSTEM TRANSFER VECTOR

This is a portion of B-system space starting at '1E000' and running for approximately 60 locations (ie, to about '1E040') which is used as an entry portal to B-system Each location in the B-system transfer vector corresponds to one B-system call. A B-system call is made by an attempt to execute (usually, though not necessarily, by means of branch and link instruction) the corresponding B-system transfer vector location. The locations do not contain instructions, but the ensuing trap will be correctly interpreted by B-system as an attempt to call B-system.

In effect, one calls B-system by an indirect reference through the transfer vector. This method ensures that the user's programs will not have to be changed when the location of the code to implement a system call changes as the result of a modification to B-system. Also, at the time the trap is handled and identified as a B-system call, the process is given write access to B-system space. Before returning to user code the B-system requests that the A-system return B-system space to read/execute status. This protects B-system space from modification by user code.

BYTE

A byte is a unit of memory on the Sigma-7 and the virtual computer. One byte is comprised of 8 bits; each word is comprised of four bytes. The bits of a word are numbered from left to right, beginning with 0, as are the bytes in a word. Thus bytes 0, 1, 2, and 3 of a word begin with bits 0, 8, 16, and 24 , respectively.

CLOSE

This is the operation of severing the association between a file and a keyword which was established by a B-system OPEN or ENTER call. A keyword is closed by executing the B-system CLOSE call. Closure of a keyword results in the loss of all access to the file which was open in that keyword; any pages of the file which were COUPLED using the closed keyword immediately become uncoupled. Keywords 14 and 15 ('e' and 'f') may never be closed by a normal user, as they are reserved for B-system use. A keyword may also be closed during the operation of the B-system calls open, ENTER, and CREATE. Closure of a keyword is sometimes spoken of as closure of the associated file.

Files which are no longer open in any keyword are subject to destruction by the garbage collector when (and only when) all directory entries pointing to the file have expired.

CONSOLE

(see GRAPHICS CONSOLE)

COMPUTATION FILE

There is one computation file for each job. It contains the sub-pacts for every process belonging to the job. Each job is assigned a computation file when the job is created and that file is entered in the job's root directory under the name USERCOMPF. The user is normally uninterested in this file, and cannot write into it.

COUPLE

Coupling is the act of logically associating a file page and a virtual memory page frame. A file page is made accessible to a process by coupling it to a virtual page frame in the process' virtual address space. The file containing the file page must be open in some keyword of the process at the time of the couple operation and the coupling between the file page and the page frame is automatically broken if the keyword is closed. A page frame in virtual memory may not be successfully referenced until some file page has been coupled into it (although this does not guarantee a successful reference). The act of coupling is performed by executing the B-system COUPLE call. This B-system call must be made for each separate page frame - file page association desired. Once a file page is coupled into a page frame, any

reference to the page frame is a reference to the file page and every change to the page frame is a change to the file page also. The same file page may be coupled simultaneously into many different page frames in the same or different processes. Only one file page may be coupled into any one page frame at a given time. File pages may be uncoupled from page frames by use of special arguments to the B-system COUPLE call, or by coupling a new page on top of the page frame. Note that the couple operation provides the only access to data in files and is the primary method for sharing files. It also provides the only means of placing data into virtual memory.

CREATE

One creates a file which did not previously exist. The call to B-system includes as arguments the type of the file and whether or not the file is to have the single-user property. If a data file is requested, the maximum page number which may be referenced in the file is also supplied. The file will be created and opened in a keyword specified by the user if the call is successful. The process will have full read/write/execute access to the new file. If the keyword to be used had a file previously open in it and if the call is successful, then the keyword will be closed before the newly created file is opened into it. Since the created file did not previously exist in the system, the creating process is the only process which has access to the file. This can be used to provide virgin (ie, new) data space for re-entrant programs. If the process wishes to save the file or share it with other processes, the file must be entered after creation into some directory.

DATA FILE

Quite simply, a data file (as distinguished from a system file or directory) is capable of holding data (note that program code is data, too). Each data file consists of a header page and up to 1000 data pages. The maximum allowable data page number is specified when the file is created. When the file is created, only the header page is allocated space on disk. Other pages are added at the time they are first referenced by some process. This strategy minimizes the number of active pages for which disk space is allocated. The data within the file is arbitrary; there are no preset formats required by the system although certain formats are meaningful to many processors. The data in a file may be referenced by coupling file pages to virtual page frames in some process' virtual address space.

DELETE

This is the operation of removing an entry from a directory. A special version of the B-system OPEN call may be used to eliminate a particular entry from a directory. Once the entry is deleted, it may not be used to reference the file to which it pointed. The garbage collector releases to the free page chain the pages of any file for which there are no existing directory entries or open keywords This is reasonable, since if there are no pointers to a file, it cannot be used, and hence may sensibly be destroyed.

DESTROY

To destroy a file is to completely remove it from the Gordo system and to return its storage space to the free page chain on the disk. When all entries to a file have been deleted and the file is no longer open in any keyword, it is destroyed. All record of the file is lost to the system. There is no way for a user to force destruction of a file. Rather, destruction occurs when the last reference (ie, directory entry or keyword) to a file is eliminated. This is an automatic activity of B-system and takes precedence over all other user activities. Because each of the file pages must be zeroed for security reasons before they can be returned to the free page chain, there may be a momentary pause in system response during the destruction of a large file.

DIRECTORY ENTRY

(see ENTRY)

DISK

(see SWAPPING DISK)

DISPLAY CONSOLE

(see GRAPHICS CONSOLE)

DIRECTORY

This is a special class of file used to provide access to other files. A directory, unlike a data file, consists only of a header page. The header page is used to hold up to 100 directory entries, each of which points to another file (which, of course, may also be a directory). A process may gain access to the file pointed to by a directory entry through use of the B-system OPEN call. Such entries may be placed into the directory with the B-system ENTER call. A process must have write access to a directory to make an entry in it. Directories are liable to garbage collection. The word "directory" will be used to denote a directory, mailbox, or non-garbage collected mailbox where such usage cannot cause confusion, as most properties of these three file types are identical.

DISPLAY GENERATOR

This is the hardware device which drives the graphics consoles. The Sigma-7 display generator is an electronic device which generates from computer supplied information the necessary analog control signals to display images on the console screens. In addition, the generator mediates all interrupts from the console input devices to the Sigma-7. Each display generator can drive up to four graphics consoles. There are two such generators on the Gordo system.

DISPLAY LIST

This is the current list of items to be presented on the display screen. Each user has a display list, consisting of the items (points, lines, characters) he currently wishes displayed. A-system amalgamates these individual display lists into one system display list which drives the consoles. Changing the display list will cause the picture on the display screen to be changed. Since the display screen is refreshed from computer memory, such changes will occur in the next display sweep through memory. Such sweeps occur approximately once every 25 milliseconds.

DISPLAY SCREEN

This is the portion of the graphics console which displays graphic output to the user. The display screen is a high precision cathode ray tube with a viewing area of 10x10 square inches. From a software point

of view, it is a cartesian plane, with co-ordinates ranging from '-400' (-1024) to '3FF' (1023) in each axis. The center section of this plane, namely the section ranging from '-200' (-512) to '1FF' (511) in each axis is always visible. The display hardware can plot points, lines, or characters anywhere in the larger plane. The edges of the plane "wrap around", so that plotting a co-ordinate less than '-400', which is off the left edge of the display, moves the item back on from the right edge. Characters may also be displayed in a packed text mode, in which case the right edge of the visible section causes an automatic linefeed and return to the left edge of the visible section.

EBCDIC

(Extended Binary Character Digital Interchange Code) - This is the IBM "standard" character code. The EBCDIC character code is a 256 character set used as an IBM standard and as a de facto industry standard. Several of the XDS programs (probably all of the XDS programs with which a Gordo user is likely to come in contact) make use of this code and it is perforce implemented on the Gordo system where such programs are used. In particular, EBCDIC is used with the XDS-supplied FORTRAN compiler, the assembler, the loader, and the library. Standard translation tables are available between EBCDIC and the Gordo character set. Gordo can read cards punched in both sets.

ENTER

One "enters" a file in a directory. A process may make an entry in a directory for a file by means of the B-system ENTER call. At the time of the call, the process must have both the directory and the file open in its keywords. Once the entry has been made, it may be used by the same or other processes to gain access to the file (assuming, of course, that they satisfy all of the access conditions). All of the attributes for the entry are specified as arguments to the B-system ENTER call.

ENTRY

An "entry" is a reference to a file in a directory. Each directory consists of from 0 to 100 entries. Each entry provides access to one file, giving a name, access level, security level, and lifetime for the file. The same file may have many different entries pointing to it, each with a different set of attributes for the file. Note that the name, access, security, and lifetime attributes are properties of the

entry, not of the file. The entry is first created and placed in the directory by means of the B-system ENTER call. The file which the entry references may be accessed with the B-system OPEN call. The entry may be deleted from the directory with a special version of the B-system OPEN call. The attributes of the entry control access to the file. For example, a process running at security level 2 cannot open a file from an entry which has a security level of 3. An entry may be deleted by the Garbage Collector when its lifetime has expired. Entries in mailboxes and non-garbage-collected mailboxes also contain a destination job number and only a process running under the destination job may open a file from such an entry.

FILE

A file is an object in the Gordo system in which information may be stored. Files are used to maintain all user programs and data. Each file consists of 1 or more pages numbered from 0 up. No file may have more than 1001 pages (numbered 0 - 1000). Page 0 (the header page) is always pre-empted by the system for maintenance of accounting information for the file. The user may read but not write page 0. When a file is initially created, only page 0 exists in the file and other pages are added as they are explicitly coupled for the first time. Each new page is initialized with zeroes. The user may subsequently enter his own data into the file. Three special classes of files (directories, mailboxes, and non-garbage-collected mailboxes) are used to build and maintain collections of files. There are a number of B-system calls which manipulate files. Among them are CREATE, OPEN, ENTER, COUPLE, and CLOSE. A user may have two files with the same name at the same time in his private directory (for example, the root directory). In the case of an ordinary directory, the file most recently created is the one referenced; ordinary directories are operated as stacks. Non-garbage-collected mailboxes, on the other hand, are maintained as queues.

FILE NAME

A "file name" is a ten character string, used to identify a file. Any ten character string from the Gordo character set may be used to identify a file, although some may be ill-advised (eg, the string of ten blanks, or names containing a period). Each directory entry contains a file name for the file to which it points (note that the file name is an attribute of the entry and not of the file itself). Any attempt to open the file through a directory entry must present a file name which

matches that in the entry. Thus, files are referred to by the file names in the entries which point to them.

FILE PAGE

A file is (logically) a consecutively addressed block of some number of words "n", numbered 0 through (n-1). Each file is divided into 512 ('200') word sections called pages, numbered 0 through ((n-1)/512). A Gordo file must be an integral number of pages in length. File pages cannot be shared between files. Gordo maintains a supply of unused file pages on the disk which may be used to supply file page requests. This supply is replenished every time a file is destroyed.

FORK

This is the act of creating a new job or process. New jobs or processes are created by means of the B-system FORKJOB and FORKPROC calls. Any process may fork a new brother process (which is attached to the same job), but only special system processes (in particular, those of the logger) may fork new jobs. The newly created job or process may be allowed access to some of the files of the creator.

FUNCTION BOX

A "function box" is a 4x4 array of lighted pushbuttons; one function box is attached to each console. It is one means of supplying console input. Each button may be off or on, and when on a button is illuminated. The user has complete control over the buttons, turning them on and off at will. The top two rows of buttons will remain latched on if depressed from the off position and must be depressed again to be turned off. the bottom two rows spring back to the off position as soon as they are released. Each button change of state generates a service request. In addition, there are plastic overlays which slip into the function box housing and which may be used to name the buttons. The presence or absence of an overlay can be sensed and each change of overlay generates a service request. The function box is used as input device for binary decisions within a program.

GARBAGE COLLECTOR

The "Garbage Collector" is a system job which keeps the disk free of unaccesable files. The Garbage Collector periodically sweeps through the entire Gordo file structure looking for directory entries whose lifetimes have expired. When such an entry is found it is deleted from its host directory. This action, of itself, does not clear the disk of an unused file. However, if the entry was the last one pointing to its file, and there do not exist any open keywords pointing to the file, then the file will be destroyed and its disk space freed. Since most files have only one entry pointing to them, this technique is effective. The Garbage Collector does not delete the entry if the file is currently open in some KEYWORD; rather, it extends the lifetime of the entry by about 15 minutes. This guarantees that files currently in use will not have their entries deleted.

GLYPH

The term "glyph" may refer to either of two entities. At the most primitive level, a glyph is a set of instructions for controlling the actions of a display generator as it produces lines and points on a display console. One creates a glyph by making a B-system "INSERT" call; the display generator instructions which are to form the glyph must previously have been placed in the I/O buffer by a "MOVEIO" call.

GORDO CHARACTER SET

This is the principle character set used on the Gordo system. Gordo makes use of a 256 character set defined by its desplay hardware. This character set corresponds closely with laboratory wide ASCII, but there are a few discrepancies, usually where Gordo has substituted a new graphic character for one of the ascii characters. This character set is used in all Gordo text files and by a number of the utility processors.

GRAPHICS CONSOLE

A "graphics console" is a remote display station for interactive graphics. A console consists of a high-precision 10"x10" cathode ray tube display screen, a keyboard capable of generating 256 distinct characters, a function box with eight latching and eight non-latching switches and changeable switch overlays, and a light pen. Points, lines

in either solid or dashed mode and characters may be displayed in any one of four intensities, in visible or invisible mode, or in blinking or stable mode. Characters may be displayed in any one of four character sizes. Items are displayed on a cartesian co-ordinate grid running from -512 to 511 in each direction on the visible screen (there is a 512 point extension in each direction for additional co-ordinate area which is not displayed). The keyboard is used primarily to input text a line at a time. The function box switches may be used for on/off decisions as the current program sees fit. The setting of the switches may be changed at any time by the user. The light pen is used to sense the presence of a display item at a particular point on the screen and is primarily used to pick out items of interest from a complicated display. Service request generators for the console include the carriage return, the light pen, the light pen in search mode, the function box keys, the function box overlay switch, the edge violation trap, and the screen wraparound trap.

HEADER PAGE

The first page of a disk file is called the "header page." It is always used for internal Gordo accounting. Its contents cannot be modified by the user. For directories, the header page also contains all information about the entries for the directory (thus a directory is always exactly one page long). The header page is seldom of any direct interest to the normal user.

HYPERAWAKE

This is one of the three scheduling states of a process. An awake process becomes hyperawake when another process directs a B-system WAKE call towards it or when its console requests service. A hyperawake process will only fall to the awake state, and not all the way to sleep, when it executes a B-system SLEEP or GIVEUP call. Otherwise, the hyperwake and awake states are the same. The hyperawake state is provided so that a process will not miss a console service request or other interrupt which arises between the time it polls the console input devices and the time when it makes a B-system SLEEP call to suspend processing.

INPUT/OUTPUT QUEUE

This is the highest priority CPU scheduling queue. Processes on the input/output queue are run before any other processes. Only I/O processes are allowed on this queue. Such processes are under hardware compulsion to process interrupts from peripheral I/O equipment quickly.

INTERACTIVE QUEUE

This is the second highest priority queue for CPU service. processes on the interactive queue run ahead of all other processes, excepting those on the input/output queue. A process is moved to the interactive queue whenever a service request is generated for the process. The process stays on the queue until one short quantum time elapses or until the process page faults. It then falls to the short quantum queue. The interactive queue is intended for very interactive programs which can service their interactions quickly in a small working set.

INTERRUPT

An "interrupt" is a software generated program interruption. A process may be interrupted by striking the interrupt key on the graphics console keyboard. An interrupt causes the process to transfer control to its interrupt handler, whose address must have been spec ified in a previous B-system SETINT call. Interrupts are handled in a manner very similar to traps.

I/O BUFFER

A 128 byte buffer is located in each process' sub-pact. The I/O buffer is the process' window to the console; all console I/O passes thru it. Most service requests (particularly reads from the keyboard) cause the buffer to be loaded with information read from the console by the service request generator. Items which are to be entered into the system display list must first be placed into the I/O buffer, usually with the B-system MOVEIO call. The I/O buffer is located at '1E31A' in B-system space.

I/O JOB

The I/O Job is responsible for the peripheral I/O devices. Because of its function, the I/O Job is given special privileges, among them the ability to execute Sigma-7 I/O instructions and the ability to make several special system calls. The I/O Job is currently job 0. It consists of 5 processes, one each for the operator's teletype, line printer, card reader, card punch, and the tape units.

I/O PROCESS

There are five I/O processes in the I/O Job. Each I/O process is responsible for one peripheral device. Currently, process 1 drives the teletype, process 2 the printer, process 3 the card reader, process 4 the card punch, and process 5 the tape units.

JOB

The "Job" is the fundamental accounting entity under Gordo. When a user logs on, a new job is forked for him. That job is given a console and is started with one process running. From then on, the user and the job are synonymous to the system. Hence a user must be logged on whenever he is running a process. The job may have any number of processes running at once, although the normal situation is for two processes to be active. The user at the console never communicates directly with the job, but only with its constituent processes. The important features of the job are its job number and its security level. The job number is used to identify files for input and output. A user will always have the same job number and thus the job number is also a user number.

JOB NUMBER

This is the number used to identify a job. Each job, when it is forked, is assigned a job number. That number will be taken from the accounting tables and is the same as the user number for the user whose log-on caused the job to be forked. The job number is primarily used to identify files on input and output. It may also be used to direct interjob B-system WAKE calls.

K

This is an abbreviation for the number 1024 = '400'. This is number is often convenient because many computer-related quantities are measured in the binary number system, and 1024 = 2**10. For example, Gordo processes have 128K of virtual memory.

KEYBOARD

There is a "keyboard" attached to each graphics console; it is the primary console input device. The console keyboard on the Sigma-7 is a large typewriter-like character input keyboard. It is capable of generating all of the 256 characters in the Gordo character set. Two of the characters are taken as process interrupt and process break keys. Software causes the keyboard to operate in a line mode (as opposed to a character mode) and each carriage return character causes an end of line condition and generates a service request. Characters are automatically echoed on the screen as they are typed, but the echoes vanish as soon as the line is accepted for input.

KEYWORD

"Keywords" are the virtual machine mechanism by which a process accesses files. Each virtual machine is equipped with 16 keywords, numbered from 0 to 15. Before a file may be used by a process, it must be open in some keyword. This is accomplished by means of the B-system OPEN and CREATE calls. Once the file is open in a keyword, pages from the file may be coupled using the B-system COUPLE call and data in the file referenced. The keyword number is an argument to the COUPLE call. Once open, the file may also be entered into directories with the B-system ENTER call. Note that since a directory is a special kind of file, it must also be open in some keyword before the ENTER call can be successful. The association between the file and the keyword may be severed by closing the keyword. This may be done explicitly with the B-system CLOSE call or it may be the side effect of some other B-system call, most commonly OPEN. Some of the keywords have special properties or conventional uses:

KEYWORD USE

0 No file may ever be open in keyword 0. This keyword is used only to delete files from directories.

1 This keyword is available for general use. However, the currently running program file is normally open in this keyword.

2-9 These keywords are available for general use.

10 This keyword is available for general use. However, the command interpretor Master usually keeps its scratch file open in this keyword.

11 This keyword is available for general use. However, the code file for Master is usually open in this keyword.

12 This keyword is available for general use. However, when a process is initially forked, the user's root directory is open in this keyword and many utility programs rely on this fact.

13 This keyword is available for general use if no calls to the B-system PRINT call are made.

14 This keyword is reserved for system use - The user's computation file is kept open in this keyword.

15 This keyword is reserved for system use - the B-system code file is kept open in this keyword.

Keywords are the central access mechanism for files under the Gordo system and must be understood for a complete grasp of the system.

LIFETIME

This is the length of time an entry is allowed to stay in a directory before it is deleted by the garbage collector. The current lifetimes are 1 hour, 2 hours, 4 hours, and infinity, coded as 0, 1, 2, and 3, respectively. When an entry is made, a lifetime is specified for it. The entry will not be liable for garbage collection until its

lifetime has expired. Every time the file to which the entry points is opened from the entry, the entry gains a new lease on life equal to its original lifetime. If the lifetime of an entry expires and the garbage collector is ready to delete the entry, the entry may still gain an additional 15 minutes grace if the file to which it points is open in any keyword. These extensions may occur as often as necessary. Note that the lifetime is a property of the entry and not of the file; the entry dies when its lifetime expires, but the file is not destroyed until all references to it have been eliminated. An infinite lifetime may only be assigned to an entry by system processes and is used primarily to protect system utility programs from garbage collection.

LIGHT PEN

A "light pen" is a pencil-shaped device, one of which is attached to each display console. Its tip is sensitive to light, and when pointed at the display screen, can report the display element at which it is pointed. Thus, it can be used to select a particular element of the display list. Under Gordo, the light pen generates a service request every time it senses light. The light pen may be inhibited, under user control, and prevented from seeing specified portions of the display list.

LIGHT PEN SEARCH MODE

This is a software-supplied mode of light-pen operation which allows the location of display screen points not in the current display list. The light pen cannot locate screen locations which are not lit by some display element. However, light pen search mode can report the x-y coordinates of such a point by very quickly sweeping the screen with a very fine grid of lines. The search is activated by a B-system SEARCH call and if the light pen is seen by the search mode, a service request is generated.

LOGGER

The "Logger" is a system job which performs accounting services. Processes of the Logger job maintain all unused consoles with a log-on message. When a user appears at the console and presents a correct identification and security combination, the Logger process tending that console forks a new job for the user, gives it the console, and retires until the user is ready to leave the system. When the user logs-off the

Logger process for his console kills the forked job, recovers the console, computes accounting information for the user, and returns the log-on message to the screen.

LONG QUANTUM QUEUE

This is the lowest priority CPU scheduling queue. processes on the long quantum queue receive last priority for CPU service, following those on the short quantum queue. A process on the long quantum queue runs for one long quantum time and then goes to the end of the queue. The long quantum queue is intended for processes which are compute bound.

LONG QUANTUM TIME

A process spends a "long quantum time" at the head of the long quantum queue before returning to the end of the queue. This time is currently '200' milliseconds, the same as the short quantum time. The system is fairly sensitive to changes in this interval, particularly with respect to console response time.

MAILBOX

This term designates a special class of files used to provide access to other files. Mailboxes operate in the same way as directories with the following three additional rules: 1) when a file is opened via a mailbox entry, it is deleted from the mailbox. 2) To open a file from a mailbox entry, the opening process job number must match the destination job number in the entry for the open to be successful. Some system and I/O Jobs are released from this rule. 3) A process may enter a file into a mailbox to which it does not have write access. Mailboxes are subject to garbage collection. They are used mainly in the system I/O file structure, although ordinary users may create and use them. The word "mailbox" may be used to mean both a mailbox and a non-garbage-collected mailbox when no confusion is possible, since the properties of the two are very similiar.

MASTER

The "Master" is a utility program provided to each process when the process is created. The master is a general purpose command interpreter and debugger. It performs most of the common utility operations for the user, such as displaying the names of files within directories, communicating with the I/O processes, preparing user programs for execution, executing B-system calls from the console, entering and deleting breakpoints in a program, and displaying on the console the contents of memory locations and registers to aid debugging. The Master is simply a program; it has no special privileges. It is maintained by the system staff, however, and is the major interface between the user, seated at a console, and the Gordo system.

MASTER PROCESS

Process 1 of each job is termed the "master process"; it is the process which the Logger creates for a user when he logs on to the system. It is initialized with a keyword for the job's computation file, the B-system, and the code file for the program Master. The Logger sees to it that the B-system and a one page bootstrap loader are coupled into the Master Process' virtual address space before it begins execution. The bootstrap loader then couples down the code pages for the program Master, which then takes control. The user then interacts with the system via Master. If he asks to execute a program, Master will first fork a new process for him, namely process 2, which is an exact duplicate of process 1. The Master of process 2 will then find the entry for the desir ed code file in the user's root directory, open it into a keyword, couple down the code pages needed by the program, and branch to the program's entry point. If the user's program aborts for any reason, control is passed by the system to the Master in process 1. Note that the user's code is run in process 2, rather than in process 1 (the Master process), so that it cannot in any way abort the Master process. If the user's program runs successfully, control will return to the Master of process 2. Execution of subsequent user programs will be done in process 2.

The Master process may make a few system calls which no higher numbered process may make. Otherwise it is identical to the other processes of the Job.

NON-GARBAGE-COLLECTED MAILBOX

This term is used to designate a special class of files used to provide access to other files. Non-garbage-collected mailboxes operate in the same way as mailboxes, with the following exceptions: first, the entries in a non-garbage-collecte d mailbox form a queue rather than a stack; secondly, a non-garbage-collected mailbox is not subject to garbage collection. Non-garbage-collected mailboxes are used by the system to provide strictly queued output (eg, first-come, first-served processing of tape requests) and to shield files waiting for output from garbage collection.

OPEN

This is the act of associating a file and a keyword. A file may be opened into a keyword by either of the B-system calls OPEN or create. Once a file is open, it may be accessed by the process which opened it. (Note the use of "open" in adjectival form. A file is said to be "open" if it has been successfully opened into some keyword.) No file may ever be referenced by a process unless it is first opened in some keyword of the process. When the file is opened, access controls from the directory entry which pointed to the file are maintained in the keyword. (A newly created file is not open from any directory, of course, and is always at full access.) While a file is open in some keyword, no entry for that file may be deleted from any directory by the Garbage Collector and the file may not be destroyed. A file may be open in several keywords at the same time. After a file has been opened, it may then be entered into other directories from the keyword.

OUTPUT REQUEST MAILBOX

These are non-garbage-collected mailboxes maintained by the system to service output requests. A process requests peripheral output by entering the appropriate files into the output request mailbox for the desired peripheral device. In most cases, it is sufficient to enter the file being output into the mailbox, but in some cases, additional control files are also necessary.

PACT

"PACT" is an acronym for "process ACtivation Table". The PACT is used by A-system to schedule the process to which the PACT refers. The PACT of an active process is always resident in a portion of A-system. It contains all of the information necessary to allow A-system to schedule execution time for the process. Additional information necessary to actually run the process once it has been scheduled is contained in the sub-PACT.

PAGE

A "page" is a measure of memory size. One page = 512 words = '200' words. Both file pages and virtual memory page frames are one page long. Virtual memory is 256 pages long. The term "page" is often used as a contraction of the terms "file page" and "virtual page frame".

PAGE FAULT

This is the situation which arises when an executing process attempts to reference a word in a page which is not in actual memory ("resident in core"). When this occurs, the program is s uspended while A-system reads the page in from the swapping disk. This of course slows down the computation in real time, although another process may execute while the page is being fetched. Since there is some system overhead for each page fault, they are undesireable. They can be minimized by careful programming. Such effort is rewarded, for the resulting programs execute faster.

PAGE FRAME

(see VIRTUAL MEMORY PAGE FRAME)

PAGING

This is a method of dynamically allocating actual memory. Machine instructions generate 17-bit memory addresses which we call "virtual memory addresses". These do not directly refer to core memory. Instead, the address is broken into two pieces. The leftmost 8 bits constitute the "virtual page number", while the 9 bits to the right of the page number are called the "virtual word number". The virtual page

number is used as an index into a 256 byte hardware table called the "Memory Map" or "Page Table". If the virtual page number is "I", then the I-th byte of the Page Table contains the number of the core page in which the virtual page "I" is actually to be found. The virtual word number then identifies which word within this core page is actually being addresssed.

The virtual page number is also used as an index into a 256 entry "Access Table". The "I-th" entry of this table indicates what kind of access the user has to page "I" of his process. If it indicates that the page is not in core, and the user's program attempts to reference a word in page "I", a "page fault" is generated. Program execution will be suspended while the operating system brings the needed page into core from disk, and updates the user's Access and Page Tables, after which he will continue execution. (See (2) and (7) for a more detailed description of paging, and of other methods for implementing such a "virtual memory".)

PRE-LOAD

This is a mechanism wherein the A-system loads a process' pre-load chain before granting the process control of the CPU. The Gordo system attempts to anticipate the needs of a process for file pages by loading the pages in the process' pre-load chain before the process is given CPU service on the long quantum queue. This strategy will work if the pre-load chain is a good estimate of the process' working set and if there is room in swapping space for the pre-load chain. Under the current hardware configuration, many processes have working sets nearly as large as swapping space so that there is no room available to pre-load the next process in the queue. It is also possible for a programmer to write a program with a very badly behaved working set which will defeat this strategy. The planned enlargement of swapping space will greatly improve the success of the pre-load strategy.

PRE-LOAD CHAIN

This is Gordo's estimate of the pages in a process' working set. It is formed by keeping track of the last few pages referenced by the process. For a process with a well-behaved working set, this estimate will be quite good. The length of the pre-load chain is a system parameter and system efficiency is very sensitive to it.

process

A "process" is the computational entity which runs on a virtual computer. Each job consists of one or more processes, and has a process number which is unique within the job (the numbering begins at 1). Processes may be forked, killed, suspended, and restarted by appropriate B-system calls. Only one process within a job may be attached to that job's console (The I/O Job and Logger are the only exceptions). The user may think of the process as a normal computer which he transacts with through the console and the job as a network of such computers. The program Master routinely creates and kills processes while initiating and terminating program execution for the user and it is typical to have two processes active in a normal user job. Any single job is limited to 512 active processes.

PUBLIC

This is a system directory containing utility programs and other system directories. PUBLIC is the central directory of the Gordo system. It contains all of the utility programs, compilers loaders, etc., which are necessary for normal user operations. In addition, a number of output request directories and other system directories are entered in PUBLIC.

QUEUE

A "queue" is a data structure in which the first item entered is the first item removed. A queue is sometimes said to be a "first-in-first-out" (FIFO) storage. Items in a queue are serviced in the order of their arrival. The entries in non-garbage-collected mailboxes are arranged as a queue.

RE-ENTRANT

A block of code is said to be "re-entrant" if it does not modify itself, and if it does not store data local to itself. The concept of re-entrant code is best understood by counter-example. Suppose that two processes attempt to execute code from a shared file which saves the value of a temporary "T" local to itself. Suppose that within this shared code, "T" is set to 10 by process 1, which is then interrupted so that process 2 may execute. Suppose that process 2 then sets "T" to 20. When process 1 next executes, "T" will have the value 20, not 10,

as it should for process 1. Clearly, this is intolerable. But if no program can store into the shared code file, this problem cannot arise, and several processes can safely share one code file. If the code in a code file does not write into itself, then it is termed "re-entrant." This has the advantage that there need only be one copy of the code for a re-e ntrant program, even though more than one process may be using it at one time. In the context of Gordo, all processes which execute code from a re-entrant program couple its code pages into their virtual address space at read/execute (and not read/write/execute access. Since the program generally needs to store information somewhere, the program will typically create a scratch file for this purpose. Each process will then use the same copy of the code file, but a separate scratch file for data storage. The current FORTRAN compiler does not generate re-entrant code; the XPL compiler does.

ROOT DIRECTORY

There is a "root directory" for each user; it contains his private files. At log-on time the Logger passes the user's root directory to the Master Process which it creates for the user. (Of course, a private root directory with no private entries will not be allocated disk space.) Every process is created with keyword 12 open to the job's root directory. The directory contains entries pointing to the PUBLIC directory and the job's computation file ("USERCOMPF"). The root directory is used for storage of the user's private files. Most utility programs search the root directory for all of their working and source files.

SEARCH STRATEGY

When a file is opened from a directory, one of the arguments to the B-system OPEN call is a file name to be matched against a directory entry. The directory is searched for an entry with a file name matching that specified. When such an entry is found in the directory, the other access conditions (security level, access level, single-user property, destination user number) are checked. If the entry passes all of these tests, the open is successful. If the matching entry cannot be found or a test fails, the open is unsuccessful. Since there may be more than one entry with the same file name, the order of search is important. Directories and mailboxes are searched as if the entries formed a stack and non-garbage collected mailboxes are searched as if the entries formed a queue. Thus a user's root directory may contain two files with the same name at the same time. The most recently created entry will be

the first accessed.

SCHEDULING QUEUE

A process desiring CPU service is entered in one of five queues. These "scheduling queues" determine the order in which process are allotted execution time. In decreasing order of priority these queues are: the input/output queue; the interactive queue; the short quantum queue; the long quantum queue; the timer queue.

SCHEDULING STATE

A process may be in one of three "scheduling states": asleep, awake, and hyperawake. A sleeping process requires no CPU service, as its computation is suspended. It may be found on the timer queue or the I/o queue. An awake process is performing normal computation, and is to be found on either the interactive queue, the short quantum queue, or the long quantum queue. A hyperawake process is also performing normal computation, but it may not be put to sleep by any single action by the process. The scheduling state may be lowered by the B-system SLEEP call, which the process itself issues. The state may be raised by B-system WAKE call coming from another process or by a service request generated from an attached graphics console. Each such change of state raises or lowers the scheduling state only one level, so that a hyperawake process may not put itself to sleep with just one B-system SLEEP call. Processes which are awaiting console input normally suspend themselves through the use of B-system SLEEP calls, relying on console service requests to reawaken them.

SECURITY LEVEL

This is a specification of the security required to protect data in a file. The security levels range from 0 (unclassified) to 7 (most secret). Some special levels are 2 (protect as restricted data), 3, and 5 (secret restricted data). Each user must specify a security level when he logs on to Gordo (the default level is 2). All processes within that job then operate at that security level. A process may not open a file from an entry with a security level higher than the process' own and may not make a new entry at a security level lower that the process' own. Files brought into the system by the I/O Job are given a security level by that job and files leaving the system via the I/O Job cause the operator to be notified of their security level as they leave. The

operator may hold up output of high security files to prevent disclosure of unauthorized information. Note that although we often speak of the security level of a file, this is actually a property of an entry for the file and not of the file itself. The same file may be in the system simultaneously at several different security levels because it is entered in several directories, some of which are at different security levels.

SERVICE REQUEST

A "service request" is a notification by an i/o service request generator that a device requires attention. For a normal process, the service request generators include the carriage return key on the keyboard, any change of state in the function box, a light pen hit, a light pen hit in search mode, an edge violation, or an edge wrap-around. Each generator is independent of the others. If a particular generator has no request pending, then it will generate a new service request when the user fulfills the condition for the request generator (for example, by typing a carriage return). When the request is generated, the process to which the generator is attached is awakened and placed on the interactive queue. Until the request is serviced, no other requests from that generator may be produced. A service request is cleared with a B-system call (there is one for each service request generator). At the time the request is cleared, information about it (eg, the line typed with the carriage return or the x-y co-ordinates of a light pen hit) is placed in the process' sub-PACT and is available to the process. In the case of the keyboard service request, no further lines may be input until a pending one is cleared. For the other service requests, all information generated by the device between the action causing the request and the action immediatedly preceeding the B-system call is lost. However, this first and last information is kept and reported. Service requests are similar to interrupts, since they cause their target processes to be awakened and given high priority, but they do not cause an automatic transfer to a handler routine. Rather, the process must poll all of the request generators (or at least the ones in which it is interested) to find the generator which caused the wakeup.

SHORT QUANTUM QUEUE

Processes on this scheduling queue have third highest priority for CPU services. They are serviced after the I/O queue and the interactive queue. Processes reach the short quantum queue by falling off the

interactive queue. Once there, they run until one short quantum time has elapsed. Each time the process page faults, it is moved to the end of the short quantum queue. Once its time on the short quantum queue has run out its is moved to the top of the long quantum queue. The short quantum queue is intended for processes which need to build their pre-load chains before going into a long computation and for those processes which need more time than allowed by the interactive queue to process an interaction, but are not really compute bound.

SHORT QUANTUM TIME

This is the amount of CPU time a process may use while on the short quantum queue before falling to the long quantum queue. This time is currently '200' (512) milliseconds. The system is fairly sensitive to changes in this time. Console response is particularly effected.

SHOT

A "shot" is the maximum length of time a process is allowed to run before it is stopped and another process given CPU service. A shot on the interactive queue is a short quantum time or until the first page fault. On the short quantum queue a shot is one short quantum time; one the long quantum queue it is one long quantum time.

SINGLE USER PROPERTY

A file which possesses this property is accessible to at most one user at any one time. When a file is created, it may be given the single user property. Such a file may be open in at most one keyword at any one time, although there may be many entries pointing to the file. If the file is open in a keyword, any further attempt to open it will fail until the file is closed.

STACK

A "stack" is a data structure in which the last item entered is the first one removed. A stack is thus a last-in-first-out (LIFO) data structure. Items in a stack are serviced in the reverse order of their arrival. The entries in directories and MAILBOXES (though not in non-garbage-collected mailboxes) are treated as a stack.

STATE VECTOR

Each process possesses a "state vector" (sometimes also referred to as a "context block") which contains all of the information needed to run the process. The state vector of a Gordo process is maintained in its PACT and sub-pact. The user is not normally interested in most of this information. (Indeed, he cannot reference the PACT in any way.) However, the sub-pact does include storage for the program counter, virtual machine registers, keywords, and Process memory Table (pmt), the current job and process numbers, security level, trap information, and buffer storage for console input. Note that the state vector does not include the virtual address space itself, but does contain the PMT which defines the virtual address space. The state vector completely defines the state of the virtual computer on which the process runs. This information is sufficient to run the actual computer when it next receives CPU service. A "state vector" defines a virtual computer; a "state vector" together with the virtual memory it defines constitute the process.

SUB-PACT

This is an abbreviation for "Sub-Process Action Table". Each process has a sub-pact. This table contains most of the information needed to run the process and correctly account it. The PACT and the sub-pact together form the state vector of a process. The sub-pact is one page in length and is always coupled in virtual memory between '1E000' and '13FFF' in B-system space. A user program may read, but not write, the sub-pact. The sub-pact is the only data page from any file which must be in actual memory while a process runs. Some of the information in the sub-pact is of interest to the user, particularly I/O buffers which report information about the console, and several entries which describe traps when they occur.

SWAPPING DISK

This term refers to the three million byte disk unit attached to the Sigma-7. It is used by the system to store user files. Since all user data is maintained in such files, the disk is a central part of the system. The efficient transfer of information to and from the disk is an important factor in system efficiency. Users may improve disk transfer efficiency for their programs by techniques discussed in other sections of the Gordo Manual. Users cannot directly direct the disk.

SWAPPING MEMORY

(see SWAPPING SPACE)

SWAPPING SPACE

This term refers to the portion of actual memory which can be used to run user processes. Swapping space is that portion of actual memory left over after A-system needs (currently 22K words) have been satisfied. This space is used to hold the file pages which user processes require for computation. A-system maintains this space and the mapping between file pages, swapping space, and each process' virtual address space. The mapping is dynamic and changes quickly as the needs of the system change. In particular, processes whose working sets are much smaller than swapping space will run considerably faster than those whose working sets are nearly the size of swapping space. Further, processes with working sets bigger than swapping space run very slowly, since such processes continually require transfer of file pages to and from the disk.

SYSTEM CALL

(see B-SYSTEM CALL)

SYSTEM JOB

"system jobs" are jobs which are allowed certain privileges in order that they may perform special system functions. A system job is not held to all of the requirements of access, security, and destination identification matching in file manipulation and it may create directory entries with infinite lifetimes. System jobs may be allowed access to some special B-system calls and may be allowed to execute Sigma-7 i/o instructions. Current system jobs include the Garbage Collector, the Logger, and the super-user job used for on-line system maintenance and control.

TIMER QUEUE

The "timer queue" contains all processes which have suspended processing for a specified length of time. Such processes are said to be "asleep". When that time has elapsed the process is the process is

placed back on the queue from which it came when it suspended processing. A process on the timer queue requires no CPU services.

TRAP

A "trap" is an abnormal program termination which occurs because of some abnormal condition which arises during the execution of a program on its virtual computer. The occurence of a trap suspends the execution of the instruction causing the trap and transfers control to the trap decoder within B-system. B-system diagnoses the cause of the trap and leaves that information available in the sub-pact. A transfer is then made to the process' trap handling routine which may attempt to fix the abnormal condition and continue execution of the program. The trap handler is simply a piece of program within the process' virtual memory. The B-system call SETTRAP informs B-system of the location of the trap handler and can be used to switch trap handlers during execution. Some common traps are stack overflow or underflow during execution of a stack instruction, division by zero, and attempting to reference a virtual memory word at an illegal access level (eg, attempting to write a word in a page coupled at read-execute access). Traps on the virtual computer are very similar to the hardware traps on the Sigma-7.

USER NUMBER

A unique number is assigned to each user to identify his input and output files; this is his "user number". This number is assigned at the same time log-on identification and security passwords are assigned. The user number serves also as the job number designating the master Process forked by the Logger when the user logs on. For all practical purposes this job number and the user number are identical.

VIRTUAL COMPUTER

Each process runs on a separate "imaginary computer" supplied by the system. Each of these "virtual computers" is similar to a hardware Sigma-7, but there are some significant differences. The virtual computer has '20,000' = 131,072 = 128K words of virtual memory, 16 general registers, and 16 keywords. The virtual computer has an instruction set identical to that of the Sigma-7. Some of the instructions are not allowed the normal user, just as some hardware instructions (particularly I/O instructions) are privileged on the

Sigma-7. The top '2000' = 2K words of virtual memory are pre-empted as B-system space, but the user is free to make use of the rest of virtual memory as he sees fit. Once a virtual memory page frame has been coupled to a file page , it functions just like actual memory. The traps and interrupts of the virtual computer operate somewhat differently than those on the Sigma-7, although all Sigma-7 traps have corresponding virtual traps. The main difference is that A-system and B-system intercede between the occurence of a trap and the notification of its occurence to the user program. Programming the virtual machine is almost identical to programming the actual Sigma-7. A major difference lies in the i/o programming, since the virtual machines transact all i/o with disk files by coupling (disk) file pages into virtual memory page frames.

VIRTUAL MACHINE

(see VIRTUAL COMPUTER)

VIRTUAL MEMORY

"Virtual memory" is the memory with which the operating system provides each virtual computer. Thus it is the memory which each process appears to have available. Virtual memory consists of 256 virtual memory page frames, each one page in length. Each virtual memory page frame must be associated with a file page by means of the B-system COUPLE call before it may be referenced. Once such an association has been made for a page frame, addresses within the frame may be referenced as if they were ordinary Sigma-7 memory (and are liable to the ordinary Sigma-7 access traps). The top 10 virtual page frames are pre-empted for B-system use. Also, virtual memorry addresses 0 - 15 are not available, since any memory reference within this address range will be interpreted as a reference to the general registers (again, as on a hardware Sigma-7).

VIRTUAL MEMORY PAGE FRAME

tvirtual memory is divided into '100' = 256 page sized "virtual memory page frames," numbered from 0 to 'ff'. The words in a page frame are numbered from 0 to '1ff'. Each virtual page frame may be coupled to at most one file page at any one time; no virtual memory word may be referenced unless it lies within a virtual page frame which has been couple to a file page. The memory reference may, of course, still be

unsuccessful if the page is not coupled at a sufficiently high access level.

VIRTUAL PAGE FRAME

(see VIRTUAL MEMORY PAGE FRAME)

WORD

A "word" is the basic unit of memory on the Sigma-7 and virtual computers. One word is 32 bits long. Unless otherwise stated, all memory sizes are in words. Memory is normally addressed in words. A word consists of four bytes or two halfwords. A word is one half of a doubleword.

WORKING SET

Given some (small) unit of time "tau", a process' "working set" is the set of pages which the process accessed in the last "tau" time units. "Tau" is a parameter which for the purposes of the Sigma-7 may be taken to be one shot. The smaller a process' working set, the fewer the number of pages which must be transported to and from the disk as the process is started and stopped by the CPU scheduler. Also, Gordo attempts to estimate the pages in the working set by keeping track of the pre-load chain. A small working set will make this estimation more efficient with a consequent improvement in system efficiency. An abrupt change in the composition of the working set causes overhead since it requires the transfer of many pages to and from the disk. A well-behaved process will try to keep the working set small, and will vary the composition of its working set slowly and/or only occasionally.

APPENDIX 2

B-SYSTEM CALLS

TABLE OF CONTENTS

[SU => call may be made only by System-User.]
[MP => call may be made only by the Master Process of a job.]
[* => call may be invoked by the Master CALL command.]

1E022	LPEN		Check lightpen interactions	#22
1E023	EDGE		Check edge-wrap-around interactions	#23
1E024	MKDN	SU	Destroy references by directory	#24
1E025	FINPROC		Simulate CTRL-+	#25
1E026	SETINT		Set interrupt location	#26
1E027	SERVICE		Call master for service	#27
1E028	RESET	MP	Reset ~pdp bit for a process (in a job)	#28
1E029	STARTP	MP	Start a process (in a job)	#29
1E02A	SETPSD	MP	Set ~psd for a process (in a job)	#2a
1E02B	ASSIGN	SU	Assign console (to a job,process)	#2b
1E02C	ASSIGNP	MP	Assign console to a process (in a job)	#2c
1E02D	FINPROC		Simulate CTRL-+	#2d
1E02E	POINT		Use lightpen pointing	#2e
1E02F	FORKQ	SU	Fork a process (in a job) on a queue	#2f
1E030	INTPROCJ	SU	Interrupt a job,process (CTRL-*)	#30
1E031	BRKPROC	M	CTRL-+ a process (in a job)	#31
1E032	BKRPROCJ	SU	CTRL-+ A JOB,PROCESS	#32
1E033	RCLOCK		Read the time-of-day clock	#33
1E034	SCLOCK	SU	Set time-of-day clock	#34
1E035	CONTROL		I/o device control	#35
1E036	STATUSX		Check function box interactions	#36
1E037	COPYX		Copy a glyph	#37
1E038	POINTX		Use lightpen pointing	#38
1E039	INSERTX		Insert a glyph	#39
1E03A	MODIFYX		Modify a glyph	#3a
1E03B	DELETEX		Delete a glyph	#3b
1E03C	EXAMINEX		Examine a glyph	#3c
1E03D	EDGEX		Check edge-wrap-around interactions	#3d
1E03E	LPENX		Check lightpen interactions	#3e
1E03F	READBSYS		Get the address of a b-system variable	136

THE GENERAL FORMAT OF B-SYSTEM CALLS

B-system is called by attempting to execute one of the locations in the b-system transfer vector. Each of these locations corresponds uniquely and immutably to one particular B-system function. For example, '1E001' corresponds to the B-system CREATE call. Normally, transfer to B-system is made by executing a virtual computer "branch and link" (BAL) instruction, but any method which has as a result the attempted execution of a B-system transfer vector location is alright. The attempted execution causes a trap in B-system (the B-system transfer vector locations are at read-only access while user code is executing), but execution will be allowed to continue if the trap address is within the limits of the B-system transfer vector. After completion of the call, return will be made to the address contained in virtual computer register 'e' = 14. Note that the address following a BAL,14 *14 will not be deposited in register 14 by this instruction.

B-system calls may require zero, one, or two arguments. Input arguments are passed in virtual computer registers 'D' and 'C' (13 and 12), with only register 'D' used if only one argument is required. Similarly, calls may return zero, one, or two values directly and these are returned in the same registers, with register 'D' containing the returned value if only a single value is returned. Many B-system calls in fact require arguments which are larger than two words and thus cannot be passed in the two argument registers. In such cases, at least one argument is a pointer to a data block containing the remainder of the argument data. The exact format of this pointer varies from call to call. However, in every case the argument data block must reside in the calling process' virtual memory outside of B-system space and no part of the block may be in the virtual computer registers.

Similarly, the values developed by B-system calls are often too large to return in the value return registers. In such cases, register 'd' usually contains a success/failure indicator and the remainder of the returned information is to be found in the process' Sub-PACT. In particular, many calls return values in the I/O buffer. These calls include all of the graphics console input device interrogation calls. There is a general return value of 'ff' (255) which indicates that the attempted call was not within the B-system transfer vector This value is returned in register 'd'. The condition code is set upon return and may be tested for the presence of a positive, negative, or zero value in register 'd'.

The Format of B-system Calls

Each B-system call is separately described. The sections of the description are as follows:

- Title line. Mandatory. The name of the call and a one line description of its function.

- Entry point. Mandatory. This is the location in the B-system transfer vector whose attempted execution will invoke the call.

- General Description. Mandatory. This is a (concise) one paragraph description of the call.

- Argument Register Setup. Optional. This is a description of the argument registers used and the values required therein. If a register is broken up into fields, the name and bit specification of each field is supplied. If the call may be invoked from the console, the arguments expected by Master and the order in which they are typed are indicated.

- Auxilliary Argument Data Block. Optional. This is a description of any auxilliary data areas needed as arguments. The format of the description is the same as that for the argument register setup section.

- Detailed Specification. Mandatory. This is a detailed description of the operation and effects of the call.

- Returned Values. Optional. This is a description of the values returned (if any) in registers 'd' and 'c'.

- Side Effects. Optional. This is a description of any side effects caused by the call which may be of interest to the user.

- Programming Notes. Optional. Miscellaneous information which may be of use to the user, and possibly an example of how the call might be made, may be placed here.

Mandatory sections must appear in the description. Optional sections are omitted if their contents are irrelevant to the call (for example, the Returned Values section is ommitted if nothing is returned).

Bit specifications are given in the following way: "M-N(P)", where "M" is the starting bit of the field, "N" is the ending bit, and "P" is the number of bits. Within any argument or value, the bits are numbered from zero on the left. For multi-word arguments, bit numbering continues without break across word boundaries.

COUPLE

ENTRY: '1E001'

This call may be invoked from the console.

GENERAL DESCRIPTION: The B-system COUPLE call is the mechanism by which file pages are associated with virtual memory page frames. Thus it is the primary tool by which the Gordo programmer defines the contents and structure of a process' virtual memory.

ARGUMENT REGISTER SETUP: Only register 'D' is used by COUPLE. Its fields are as follows:

AAUUKKKK VVVVVVVV FFFFFFFF.FFFFFFFF

Arg#	Name	Bits	Use
A	ACCESS	0-1(2)	The access field is a partial specification of the access level at which the page is to be coupled.
U		2-3(2)	unused.
K 1	KEYWORD	4-7(4)	The "KEYWORD" field specifies the keyword in which the file whose page is to be coupled is open.
V 2	VPAGE	8-15(8)	The "VPAGE" field specifies the virtual memory page frame into which the couple is to be made.
F 3	FPAGE	16-31(16)	The "FPAGE" field specifies the page number of the file page to be coupled.

DETAILED SPECIFICATION: COUPLE attempts to couple file page "FPAGE" from the file open in keyword "KEYWORD" into virtual memory page frame "VPAGE" at an access level partially specified by "ACCESS". The steps of this process are as follows:

1: VPAGE is checked to see if it names a page in the B-system space (ie, is "VPAGE" greater than 'f0'?). If it does, an error return is made. B-system may makeCOUPLE calls to itself, in which case this test is not performed.

2: A check on the value of "KEYWORD" is made.
   a: If "KEYWORD" = 0, the virtual memory page frame VPAGE is unconditionally uncoupled from any page coupled to it. Control is then returned to the caller.
   b: If "KEYWORD" > 0, keyword ""KEYWORD"" is checked to see if any file is open in it. If no file is open, an error return is made.

3: The file type is tested. If the file open in keyword "KEYWORD" is a directory, the value of "FPAGE" must be zero (for only page zero exists in a directory). Otherwise, an error exit is taken.

4: FPAGE is checked against the maximum page number allowed for the file (which was set when the file was created). If "FPAGE" is greater than that maximum, then an error exit is taken.

5: If the file page "FPAGE" has not yet been referenced and a new page is needed from the free page chain maintained by Gordo, a request is made for such a free page. If no free pages exist, an error exit is taken.

6: An access level calculation is made. The access level for the virtual memory page frame is the minimum of "ACCESS", the access level for the entire file as specified by the keyword "KEYWORD", If "FPAGE" = 0, then the page will be set at not more than read-only acces s.

7: The file page "FPAGE" is coupled into virtual memory page frame "VPAGE" with the access calculated in step 6. A successful return is made.

RETURNED VALUES: Only register 'D' is used. The values are:

0 Successful completion of the call.

1 ERROR - There is no file open in keyword "KEYWORD".

2 ERROR - The file open in keyword "KEYWORD" is a directory and "FPAGE" is not 0.

3 ERROR - The virtual memory page frame requested is in B-system space (ie, VPAGE > 'F0').

4 ERROR - The file page number "FPAGE" is greater than the maximum file page allowed for the file, or else there are no more free pages available from the disk and this couple would require the assignment of a new file page.

PROGRAMMING NOTES: In assembly language a typical call to COUPLE might look like

```
.
.
*
*   DEFINE A DATA AREA FOR THE ARGUMENT
*
COUPARG   GEN,2,2,4,8,16     1,0,5,X'23',X'44'
.         .
.         .
          LW,R13             COUPARG
          BAL,R14            X'1E001'
          BNEZ               COUPERR
.         .
.         .
```

This call would attempt to couple file page '44' from the file open in keyword 5 into virtual memory page frame '23' with access level no greater that read/execute. If the couple is unsuccessful, a branch is made to COUPERR. A call to COUPLE with "KEYWORD" = 0 can be used to uncouple a page from a virtual memory page frame. In fact, any successful couple uncouples any file page previously coupled into the target virtual memory page frame.

CREATE

ENTRY: '1E002'

This call may be invoked from the console.

GENERAL DESCRIPTION: The B-system CREATE call attempts to create a completely new file with the attributes defined by the calling argument. Access to the new file is available to the creating process only until that process takes affirmative action to share the file.

ARGUMENT REGISTER SETUP: Only register 'D' is used for input. Its fields are as follows:

AAUUMSSS TTTTKKKK PPPPPPPP.PPPPPPPP

	Arg#	Name	Bits	Use
A	1	ACCESS	0-1(2)	The "ACCESS" field specifies the maximum access with which the file may ever be referenced. ***** IMPORTANT ***** This argument appears for historical reasons only. Under the current implementation, this argument is completely ignored.
		U	2-3(2)	unused.
M	2	SINGLE.USER	4(1)	If "SINGLE.USER" = 1, the created file will have the single-user property and may be open in at most one keyword at any one time.
S	3	SECURITY	5-7(3)	The "SECURITY" field specifies the minimum security level at which the file may ever by referenced. ***** IMPORTANT ***** This argument appears for historical reasons only. Under the current implementation, it is completely ignored.
T	4	FILE.TYPE	8-11(4)	The file type specifies the kind of file to be created. The codes for file types are given in the following table

	CODE	TYPE
	0-6	Data files.
	7	Non-garbage collected mailbox.
	8	Directory.
	9	Mailbox.
	10	Computation file (system use only).
	11-14	Data file.
	15	System file (system use only).

K 5 KEYWORD 12-15(4) The "KEYWORD" field specifies the keyword into which the new file is to be created. If a file is open in this keyword at the time of the call, it will be closed before the new file is opened into the keyword.

P 6 MAXPAGES 16-31(16) The "MAXPAGES" field specifies the maximum page number which may be referenced in the file. This argument is irrelevant for any kind of directory.

DETAILED SPECIFICATION: CREATE attempts to create a new file (of type given by the code in "FILE.TYPE") into keyword "KEYWORD". The file will have the single.user property if "SINGLE.USER" = 1. No page number greater than "MAX.PAGES" can ever by referenced in the file. The "ACCESS" and "SECURITY" fields are ignored in the present implementation. If any file was open in keyword "KEYWORD" at the time of the call and the call is successful, the file previously occupying keyword "KEYWORD" will be closed before the creation of the new file. The steps in this process are:

1: If "KEYWORD" = 0, make an error return.

2: If "KEYWORD" points to a system file, make an error return.

3: If "FILE.TYPE" selects a non-garbage collected mailbox and the calling process is not a system process, make an error return.

4: If "FILE.TYPE" selects a computation file and the calling process is not a system process, make an error return.

5: Attempt to allocate a free disk page for the file header. If no free pages are available, make an error return.

6: Close any previous file occupying the keyword.

7: Fill in the keyword and return successfully.

RETURNED VALUES: Information is returned in register 'D' only. Its values are as follows:

0 Successful completion of the call; the file was created.

12 Error - The keyword specified is 0 or is reserved for system use.

13 Error - There is no disk page available for the file header.

14 Error - The calling process requested a file type which it is not authorized to create.

PROGRAMMING NOTES: In assembly language, a typical call to CREATE might look like

```
        .
        .
        *
        *  DEFINE THE ARGUMENT FOR CREATE
        *
CREATEARG   GEN,2,2,1,3,4,4,16   0,0,0,8,4,0
        .           .
        .           .
                    LW,R13              CREATEARG
                    BAL,R14             X'1E002'
                    BNEZ                CREATERR
        .           .
        .           .
```

This call attempts to create a directory (without the single-user property) into keyword 4. If an error occurs, control will be transfered to "CREATERR". Note that the "MAX.PAGES" field is irrelevant in a call which creates any kind of directory and that the "ACCESS" and SECURITY" fields are ignored in this implementation.

The file created by a call to CREATE is accessible only to the creating process. Until that process shares the file by some affirmative action (by entering the file in some directory or by forking a job or process), no other process has any access to the file. Thus, such files can be used to supply scratch data space for construction of re-entrant programs. When the keyword is closed the file space will be released.

READBSYS

Entry: '1E03F'

GENERAL DESCRIPTION: This system call allows the user to obtain the address of a B-system variable whose value he may find useful. While a user may directly read the contents of any B-system variable whose address he knows, it cannot be guaranteed that any variable will be at the same address in succeeding versions of B-system. This call therefore allows the user to write programs which obtain the address of a B-system variable at run time.

ARGUMENT REGISTER SETUP: Only register 'd' is used for input. It contains an integer indicating which variable is of interest. The codes currently available, and the variables which they designate, are as follows:

0 (address of) "DISCPGS", the number of disk pages currently allocated.

1 (address of) "DISCTOP", the page number of the last page on disk (the total number of pages on the disk is "DISCTOP"+1).

2 (address of) "BYTECNT", which contains a count of the number of bytes in the I/o buffer "IOBUFR".

3 (address of) "IOBUFR", the 128-byte i/o buffer.

4 (address of) "JOBNO", the number of the job under which the process is r unning.

5 (address of) "PROCNUM", the number of the process in which the caller is running.

6 (address of) "TIMELFT", the number of milliseconds which the process has left to run. (This variable is currently ignored by the system.)

7 (address of) "FREPAGES"

RETURNED VALUES: Information is returned in register 'D' only. If the code specified as input in register 'D' is invalid then 'D' will be returned with the value -1. If the code specified is valid, then 'D' will be returned with the word address of the associated variable.

PROGRAMMING NOTES: In assembly language, a typical call to readbsys might look like:

```
        .
        .
READBSYS  EQU         X'1E03F'
#DISCPGS  RES         1
        .
        .
        LI,R13    0           CODE FOR NUMBER OF DISK PAGES
        BAL,R14   READBSYS    B-SYSTEM CALL
        BLZ       ERROR
        STW,R13   #DISCPGS    SAVE ADDRESS OF VARIABLE
        .
        .
        LW,R2     *#DISCPGS   GET # OF DISK PAGES NOW ALLOCATED
        .
        .
```

This call obtains the address of the B-system variable"DISCPGS". The address is saved in the local variable "DISCPGS". At a later time the number of disc pages allocated is obtained by loading indirectly through "#DISCPGS".

APPENDIX 3

PROGRAMMING STANDARDS

"indeed, one of my major complaints about the computer field is that whereas Newton could say, 'If I have seen a little farther than others it is because I have stood on the shoulders of giants,' I am forced to say, 'Today we stand on each other's feet.' Perhaps the central problem we face in all of computer science is how we are to reach the situation where we build on top of the work of others rather than redoing so much of it in a relatively trivial way. Science is supposed to be cumulative, not endless duplication of the same kind of things."

– Richard E. Hamming (3) –

There is no question but that it is harder to write good programs than bad ones. But it is also true that a good program is easier to debug, modify, and read. Good programming, like any other creative activity, requires a sense of aesthetics, and a measure of self-discipline. Nevertheless, the returns are very great.

The first and most important principle is that a good program be easy to read. We shall discuss below how this may be accomplished, but most of what we say will be derived from this rule. It is unfortunately true that computer programming is rarely taught. One cannot learn to write good English without studying the prose of accomplished writers, or having one's own efforts criticized. Yet hardly ever is the novice programmer given examples of well written programs whose style he may absorb, and even less often is one person's program criticized by another. This is a catastrophe; we are surrounded by the evidence. You should seek opportunities to have your programs read and criticized by others, and to discuss the elements of good programming with programmers who have some conception of what they are.

A second principle is that it is the algorithm chosen (and often the data structures used) which pre-eminently determine the speed of a program, not the particular coding by which the program is implemented. The fastest bubble sort in the world cannot approach the speed of an ordinary heapsort, shell sort, or quicksort. It is better to chose the right algorithm than to optimize the wrong one. Further, studies indicate that programs generally spend the bulk of their time in a relatively small protion of the code (4,5,6). For example, in the sample of programs that Knuth studied (4), over 50 percent of the execution time was typically spent in less than 4 percent of the code. This suggests that a programmer's time is best spent locating and optimizing the offending four percent. This task is greatly facilitated by compilers and systems which provide the proper kind of programming aids.(5)

Thirdly, it is often worth trading some execution speed for ease of debugging. A program which requires great effort to debug will require repeated compilation and many hours of examination. Statistics as to the average ratio of cpu time spent in compilation to execution time are unfortunately not available. One suspects, however, that for time-sharing systems (especially those with slow compilers) the ratio is typically very high. It is certain that 90 percent of the time spent in developing a program is spent on debugging - with most of the rest spent in modification. Easily debugged, clearly written programs are preferable to obscure, bug-ridden programs. A corollary is that clever coding (which should be distinguished from a clever algorithm) is hardly ever profitable. It generally results in code which is excessively

difficult to debug and modify. This is especially true of assembly language programs. If you must do something clever, document it completely.

Finally, no program is without bugs. At the time a program is designed and written, code should be included which will facilitate debugging. The execution of this code should be under the control of a switch or switches, so that it can be eliminated when the program appears to be running properly, but it should not be removed from the program; it is desireable that such debugging information can be obtained by altering a switch setting whenever the existence of a bug is suspected. Some languages offer conditional compilation (for example, the "DIF-statement" of LRLTRAN), which is very useful in this regard.

We shall now discuss a few of the stylistic devices whereby one writes clean, readable, aesthetic programs.

COMMENTS

Comments are the most important part of any program. They should be profuse and not cryptic, but to the point and clear. The program with too many comments has yet to be written. Each procedure (subroutine, function, etc.) should contain comments of the following kind:

1. At the beginning of each routine should appear a precis of the purpose of the routine and the major algorithms it implements to achieve its purpose. The name of the person who implemented the routine should be stated along with the date at which it was last modified. If more than one person has modified the routine, the date of each modification together with a brief description of the modification made should also be included. If appropriate (and available) a list of references which might assist anyone attempting to understand the program should also be included. It may be appropriate to indicate the environment for which the routine was designed, and to indicate what portions of the routine are dependant on a particular machine, library, or compiler.

2. Following this should appear a list of the formal parameters, distinguished as to input and output parameters, together with a description of the use of each.

3. Near the beginning of each routine should appear a dictionary of all non-trivial variables, describing the use of each. A similiar list of other procedures which the given routine calls, a list of those procedures which call the given

routine, or (for FORTRAN) a list of common blocks used and the procedures with whom they communicate information may also be useful.

4. Comments should appear at the beginning of each block or segment of code which performs a single logical task. A block of more than (say) 20 statements can probably be divided up into smaller blocks, each of which has its own logical function. Each such block of code should be separated from the blocks ahead of and behind it by some number of blank cards.

5. If the language allows comments on the same card as a statement, it is often useful to explain minor points in the that way.

6. The major logical divisions of a routine, such as the dictionary, data declaration, calculations, and output of debug information, should be separated in some clear and graphic manner. A line of asterisks running across the page is one such method.

7. A comment should appear at branch points (where reasonable), indicating the reason for taking each branch and what is to be done at the destination.

8. There should be a comment accompaning procedure (subroutine, function) calls, describing briefly what will be done by the procedure, particularly side effects.

9. Comments should also appear anywhere they will help to make absolutely clear what is happening.

10. Nearly every line of code in an assembly language program should be commented. Comments should be grouped, say by indentation, when this is helpful.

FLOW OF CONTROL

The overall objective is to make the flow of control as clear and easy to follow as is possible. It is very desireable that the appearance of the program listing visually suggest the flow of control, scope of do-loops and if-statements, and logical partitioning of the routine. It should be possible to see at one glance, and contain in the mind at one time, the entire flow of control throughout a routine. This is absolutely necessary if the functioning of the routine is to be understood. Indentation should be used to indicate scope. For example,

one should write

main routine then calls a number of subroutines, each of which has a clear and precise function. Depending on the size of the program, each of these subordinate subroutines may itself be subdivided yet again, etc., to whatever level is necessary. The important point is that subroutines are useful not just to avoid the duplication of code, but also allow the programmer to keep the size of each routine to a manageable (ie, understandable) size.

As the above examples also demonstrate, blanks should be used liberally to improve readability. Surrounding equal signs and operators with blanks makes it easier for the eye to recognize the elementary units (such as operators and variable names) of which each statement is composed. Variable names should be chosen mnemonically; in the case of FORTRAN cross-reference listi ngs (the "CODE ANALYSIS" of LRLTRAN) should be scanned occasionally to check for undiscovered mispellings.

If it is expected that the program will be used or maintained by others, then serious thought should be given to preparing formal documentation in addition to the program listing. This topic is discussed in (1), to which the reader is refered.

Finally, use a language appropriate to the task[*]. Compilers are best written in a language like XPL, operating systems in a language like SUE or BLISS, text editors in a language such as TRIX, numeric problems in a language such as ALGOL, PL/I, PASCAL, or ALGOL-W, and string manipulation problems in a language such as SNOBOL or TRAC. One cannot write clear, correct programs in a language which is not designed for the problem at hand; too much time is spent writing code to avoid the restrictions of the language, and this submerges the method of solution in an unpenetrable sea of detail.

---

[*] The thoughtful reader will have noted that the uniform application of this rule would eliminate the use of FORTRAN.

(This page should be replaced with a ? page insert, available with the Gordo manual and generated by "RED", which contains tables of the Gordo, Ascii and EBCDIC character sets.)