AN INTRODUCTION TO GCOS BATCH PROCESSING

1. Introduction

This manual is intended to explain the basics of GCOS8 batch processing to those who already have a working knowledge of TSS. While we will discuss the fundamental concepts of the batch world and describe the constructs you are most likely to use in batch applications, we will make no attempt to cover every batch capability in detail. For complete descriptions of various features, you should see the current copy of one of the following GCOS8 manuals:

GCOS8 OS JOB CONTROL LANGUAGE MANUAL
: for information on the various JCL control cards.

GENERAL LOADER MANUAL
: for information on the loader.

TSS TERMINAL/BATCH INTERFACE MANUAL
: for information on JRN, JOUT, SCAN, etc.

CONTROL CARDS AND ABORT CODES POCKET GUIDE
: for a handy summary of JCL.

There are also a good many "explain" files which deal with facets of the batch world. These are listed in Appendix A.

2. Batch vs. TSS

Many TSS users ask themselves why they should bother learning how to use batch. After all, a good many of the things that can be done in batch can be done in TSS too. Furthermore, TSS is an old familiar environment while the batch world can be somewhat strange and intimidating for people who are just learning their way around the system. What is batch good for and why is it better than TSS?

Batch is designed for big jobs. A big job is one that needs a lot of CPU time, a lot of memory, or both. Batch handles such jobs faster and more cheaply than TSS. TSS, after all, is an interactive system; it is designed to handle comparatively small jobs which only operate for short periods of time between interactions with the user.

Once a batch process has been started, it can run completely independently. It does not compete for shared resources with other TSS jobs, but is allocated memory and processing time outside of TSS.

Thus a big batch job will usually run faster than the same job run under TSS, especially if there are a lot of people signed on to Time-Sharing.

Another advantage of batch processing is that it allows you to do things that can't be done on TSS. For example, you can only use magnetic tapes when working in batch mode. You have more contro over the way the system schedules your job and the way your job uses peripherals. Lastly, you can have any number of batch jobs running at any one time. On TSS, you can only have one process running per terminal; even if you try to get around this by using DRUNs (detached runs), there is a limit to the number of detached processes that you can initiate. If the number of TSS users is too high, DRUNs will not run at all; batch jobs however will continue to work away despite the TSS load.

At one time, batch processes had to be run with punched cards; now however, most batch jobs are prepared under TSS using a standard text editor like FRED and then sent for batch processing via the JRN command. In this way, you can prepare a batch job with all the conveniences provided by TSS and yet run it without Time-Sharing's resource disadvantages. The process of submitting a batch job from TSS is described in Section 8.1.

3. Basic Batch JCL

In this section, we will describe the most basic Job Control Language control cards that you will need to run your batch jobs. Of course, if you are submitting your job from TSS you will not be using cards at all; instead you will be using normal disk files whose lines correspond to card images. Although it is comparatively rare these days to submit batch jobs with real punched cards, the term "control card" is often used for historical reasons.

A GCOS8 JCL control card contains four different fields, separated from each other by one or more blanks or tab characters. The fields themselves *cannot* contain blanks or tabs.

The first field of every JCL card is a "$" to indicate that it *is* a JCL card. The second field is a keyword indicating what kind of JCL card it is. The third field contains various arguments and parameters; for example, the third field of a $FILE card contains the size of the file, amongst other things. Anything after the third field is considered to be a comment. Thus a JCL card has the general form

```
$      keyword   parameters   comments
```

The "$" sign must be in column 1. Traditionally, the "keyword" begins in column 8, the "parameters" begin in column 16, and the "comments" begin in column 32. At one time these column conventions were compulsory, but these days the batch processor is versatile enough to deal with a free format. Thus the fields may begin in any column, provided they are separated from the previous field by at least one blank or tab. Note that the output listing of any batch job will show the fields aligned in the "traditional" columns mentioned above, regardless of the actual form of the JCL input.

In the text of this guide, we will usually type JCL keywords in upper case, as in $IDENT. However, most of the alphabetic characters on JCL cards may be typed in either upper or lower case.

By the time your JCL is sent to the batch processor, it should be written in BCD. If you are submitting your job from TSS with the JRN command, you need not worry about this; JRN automatically converts ASCII files to BCD, provided there are no ASCII characters in your JCL which do not have BCD counterparts (e.g. "{", "}", etc.). Thus you must make sure that the cards in your JCL stream do not contain characters which cannot be represented in BCD.

3.1 The $IDENT Card

Every batch job must have at least one $IDENT card. This card identifies the name of the batch job and the name of the user running the job. Different sites use different formats for the $IDENT card; one popular format is

```
$      ident   userid$password,banner
```

where "userid" identifies the user, "password" is the user's password, and "banner" is the name that will be printed as a header on the job's output. This is the format that will be used throughout this guide. However, it may not be the format used at your site; see your system adminstrator for your own site's format.

In the above $IDENT format, the banner will be translated into BCD, even if you submit the JCL in ASCII format. Thus the banner cannot contain any characters which have no BCD representations. Furthermore, the letters in your banner will always print out on the output listing in upper case (since BCD only has one case for alphabetic characters).

3.2 Activities

Every batch job consists of one or more *activities*. An activity is a single job step; for example, compiling a Fortran program could be one activity in a batch job while executing the compiled program would be another activity in the same job. A batch job must perform at least one activity.

There are many different JCL cards which define an activity in a batch job. In the execution report listing that is printed for each batch job you run, the JCL cards which define the different activities of the job are flagged with the letter "a" immediately to the left of the "$" sign at the beginning of the card.

Below we list some of the more common activity JCL cards.

$ FORTRAN options
: This card invokes the Fortran compiler. The options accepted by the compiler are described in the GCOS8 reference manuals mentioned in Section 1, and in "expl cc fortran".

$ GMAP options
: This card invokes the GMAP assembler. Valid options are described in the GCOS8 manuals and "expl cc gmap".

$ EXECUTE options

This card is generally used to invoke the program loader. For example, if one step in a job compiles a Fortran program, you would use a $EXECUTE card to load the compiled program into memory and execute it. For more on the $EXECUTE card, see Section 5.

$ PROGRAM core-image

Frequently used programs like compilers, assemblers, and the loader have their own special JCL cards as described above. However, most system programs (and of course all user-written programs) do NOT have their own JCL cards. The $PROGRAM card can be used to execute such programs. For example, to use TF in batch you would define the activity with a card of the form

```
$       program tf
```

We warn you that there is more to using such programs in batch than just naming the program on a $PROGRAM card. In the example above, you would need at least one additional JCL card telling the batch processor where the TF program could be found in the file system. We will say a good deal more about calling such programs in Section 6, by which time we'll have discussed the other JCL cards which are needed to make the call.

There are several other JCL activity cards in addition to the ones mentioned above (e.g. $COBOL). These are described in the GCOS8 reference manuals.

In the sections to come, we will give examples of jobs which use most of the activity cards we have discussed.

3.3 The $USERID Card

Every batch job should have at least one $USERID card. This is supplied automatically for you if you submit the job to batch through the TSS JRN command (see Section 8.1). If, however, you are submitting your job using cards, you will have to supply your own $USERID. The card has the form

```
$       userid  user$password
```

where "user" is a valid userid and "password" is that user's password; for example,

```
$       userid  alibaba$open
```

The given password is not printed on the output listing for the job.

Generally speaking, the $USERID card is placed before the $IDENT card in a batch job; this way, you don't have to specify the password on the $IDENT card (see Section 3.1). You can change userids partway through a job by placing different $USERID cards between activities in your program, but few users ever need this feature.

3.4 Snumbs

Batch jobs are identified by unique "system numbers" or "snumbs". A snumb consists of up to five BCD characters which are assigned to the job either by the system or the operators. (If the job is submitted through the TSS JRN command, JRN supplies the snumb; if the job is submitted on punched cards, the operator supplies the snumb.) The snumb for a batch job will appear at the top o the job's output listing in a line of the form

```
$      snumb   y000t
```

In this case, the job's snumb is "y000t". You may *not* specify your own snumb.

Most of the TSS commands which monitor or control batch jobs make use of a job's snumb. We will say more about this in Section 8.

Incidentally, system-generated snumbs tell you when a job was submitted to batch (e.g. when JRN submitted the job). The first character is a letter which indicates the hour of the day when the job was submitted: "a" is for jobs submitted between 0:00am and 1:00am, "b" is for jobs submitted between 1:00am and 2:00am, and so on. The next three characters are three digits indicating time within the hour in thousandths of an hour; thus "500" would indicate that the job was submitted on the half hour. The last character is "z" if the job was submitted from cards and "t" if the job was submitted from TSS. Thus a job with snumb "x250t" was submitted from TSS at 11:15pm.

3.5 The $LIMITS Card

The $LIMITS card specifies resource limits for a single job activity. Since there are default resource limits for all activities, there will be many times that you don't have to specify your own $LIMITS; however, if you are running an activity that needs more than the default amount of processor time o memory, you must inform the batch processor of the job's requirements. Once in a while, you might also want to specify a resource limit that is less than the default; this can help your job get run faster.

The general format of the $LIMITS card is

```
$      limits  time,st1,st2,lines,iotime,st3
```

"time" gives the maximum amount of processor time for the activity in hundredths of an hour. "st1" gives the initial amount of memory to be allocated for running the activity's program; this number is given in blocks of 1024 machine words or "K". For example, specifying "4K" would allocate an initial space of 4096 machine words for the program. "lines" gives the maximum number of lines of output which the activity will print to SYSOUT; this is usually specified in "K" lines. For example, specifying "4K" lines means a maximum of 4096 lines of output.

The arguments "st2", "iotime", and "st3" are not normally needed by most users; for descriptions of these arguments, see the GCOS8 JCL Reference Manual.

If any of the above arguments are omitted on the $LIMITS card, the batch processor will use the default limit for the type of activity being performed.

Below we give a simple two-activity batch job that makes use of the JCL cards we have discussed so far.

```
$       ident   alibaba$open,sesame
$       option  fortran          **to be explained**
$       fortran
        ** source for fortran program **
$       execute
$       limits  4,8K,,10K
        ** data for fortran program **
```

The first activity of the above job is a Fortran compilation. The source cards for the Fortran program follow the $FORTRAN card as shown. (Remember, if you are submitting this job through JRN, the Fortran source will automatically be converted into BCD. Since the Fortran compiler can handle either ASCII or BCD source, this is not normally a problem; however, all the letters in your program will be converted to upper case since there is no distinction between cases in BCD. You will only get into real trouble if your source contains special ASCII characters. In Section 4, we wil show how to pass your source code to the Fortran compiler from a file, thereby avoiding the problems that arise from JRN's automatic conversion.)

The second activity above executes the Fortran program. The $LIMITS card gives the Fortran program a maximum of four hundredths of an hour, an initial memory size of 8K, and a maximum of 10K lines of output (about 10,000 lines). The data for the Fortran program follows the $LIMITS card (again this data will be converted to BCD). We will discuss the $OPTION card in Section 5.1.

4. Using Files in Batch

There are a number of ways used in batch programs to describe files, and a number of JCL cards that handle such files. In this section, we will look at the basics of batch file handling.

4.1 File Codes

Every file which is used by a batch activity must be given a "file code". This is a two-character name which batch programs use to refer to files. For example, the Fortran compiler expects that the source code to be compiled will be stored in a file with the file code "s*". The GMAP assembler expects that the program to be assembled will be stored in a file with the file code "g*". Neither the compiler nor the assembler care about the actual name of their source files; all they need to know is the file code. It is up to the JCL statements of the activity to associate the file code which the program uses with the proper file.

As another example, consider a Fortran program that contains the statement

```
write(08,25) x,y,z
```

This implies a write to "unit 08". When the program is actually executed, the output will be sent to the file with file code "08". Once again we see that file codes are the way that batch programs refer

to files.

A file code can consist of any two characters that are found in the BCD character set; however, it is usually best to restrict yourself to using just letters and numbers in user-defined file codes. This will avoid naming conflicts with the special file codes used by GCOS8 system software. Most of these special file codes are identified by having a "*" as either the first or second character.

All batch compilers look for their source in file code "s*". As mentioned previously, the GMAP assembler looks for its source in file code "g*". The object decks which compilers and assemblers produce are stored under file code "b*". In the sections to come, we will discuss a number of other special file codes. These system file codes are summarized in Appendix B.

Finally, we will note that you may not have two files with the same file code connected with the same activity. After all, the program will not know what to do when it is given two files with the same file code. (If you do have more than one file with the same file code in a given activity, the system will use the one that is specified last.) However, it is possible to use the same file code in two different activities in the same job; for example, if your job contains a Fortran compile and a Pascal compile, both will have source files with the file code "s*".

4.2 Logical Unit Designators

Programs refer to files by using file codes; the JCL of a job refers to files by using *logical unit designators*. A logical unit designator is usually made up of two alphanumeric characters; the first character can be either a letter or a number, but the second character must be a number. Thus "D8", "A1", "W3" are all examples of valid LUDs.

Logical unit designators are included in the JCL to tell the batch processor which files are the same and which files are different. For example, if an output file from one activity is the input file for a later activity in the job, the JCL would specify the same LUD for the file in both activities. Thus if you were using a permanent file in several steps of a job, you would only have to provide the file's pathname the first time you used it; in subsequent activities, all you need to use is the LUD.

The LUD is a completely separate entity from the file code associated with the same file. For example, it is very common for the output file of one activity to act as input for the next activity. Since the same file is used for both activities, it would have the same LUD in both job steps; however, in the first activity the program may use the file code "ot" to refer to the file while the second program may use the file code "in". Remember, file codes are used by programs in single activities; LUDs are used by the JCL for the entire job.

4.3 Disposition Codes

Any LUD may be followed immediately by a *disposition code*. This code tells the batch processor what to do with the given file after the current activity is finished. The most commonly used disposition codes are

r

"Release" the file. This disposition code actually does release temporary files. Permanent files however are not released but merely de-accessed, in much the same way that permanent files are removed from the AFT in TSS when a program no longer needs them.

s

Save the file for subsequent activities in this job.

p

Purge the file. This is a special security disposition code. The contents of the file are overwritten with arbitrary data and the file is then released.

There are two other disposition codes that are sometimes used in connection with magnetic tapes and private disk packs.

d

Dismount the medium from the system. (This is changed to "r" if the given medium is a public disk pack.)

c

Write-inhibit the medium (e.g. take off a tape's write ring) and save the medium for a subsequent activity in this job.

When no disposition code is specified after an LUD, "r" is assumed.

Below we give some examples of LUDs with disposition codes.

x1r

The file with LUD "x1" is to be released at the end of this job step.

q5s

The file with LUD "q5" is to be saved for a subsequent job step.

h7d

The device containing the file with LUD "h7" is to be dismounted at the end of this job step.

4.4 The $PRMFL Card

The first time a permanent file is referenced in a job activity, the JCL must contain a $PRMFL card describing the file. The $PRMFL cards for the files used by an activity come after the JCL card which defines the activity (e.g. $FORTRAN or $EXECUTE), but they do not have to come immediately after. There must be a $USERID card somewhere in the job before the first $PRMFL card.

The general format of the $PRMFL card is

```
$      prmfl   fcode/lud,perm,typ,pathname
```

"fcode" is the file code that the activity's program will use. "lud" is the logical unit designator for the file; this can include an optional disposition code. "pathname" is the full pathname for the permanent file, including catalog and file passwords if the file is password-protected. If the file is

only going to be used once in the job, you do not have to specify the "lud" since the JCL doesn't need a way to remember the file after the current activity is finished.

The "perm" argument specifies the permissions with which the file is to be accessed. Some of the more common permissions are

r
: Read only access.

w
: Read and write access.

p
: Private access (read and write).

e or x
: Execute access (this is used when an activity's program resides in the given permanent file).

r/c
: Read concurrent. You should always use this permission when you are reading files that are shared with other users. For example, if you are loading a program and linking in programs from a system library, you should definitely use "r/c" rather than "r". If you just use "r", no one will be able to get at that library all the time your program is executing.

w/c
: Read and write concurrent. As with "r/c", you should use this permission when you are working with files that are shared with other users.

q
: Query permission. This is much like "r/c". The difference between the two comes when the file in question has been accessed privately by another user: "r/c" will not access a file that is currently accessed privately while "q" ignores the fact that another user has asked for the file all to himself. Thus you can still "query" a system library even if another user has been inconsiderate enough to access the library with "r" (private read) permission. Of course, there is always the chance that the library was accessed privately for a reason; for example, if you "query" the library when someone else is in the middle of updating it, you stand a good chance of obtaining garbage.

There are a number of other access permissions which can be specified on the $PRMFL card; for full information, see the GCOS8 JCL Reference Manual. If no "perm" is specified, the default is "r" (read only).

The "typ" argument on the $PRMFL card tells whether the file is random or sequential. An "r" indicates that the file is random and an "s" indicates that it is sequential.

Below we give several examples of $PRMFL cards.

```
$ prmfl g*,,s,/src.g
```
: This accesses the sequential file "userid/src.g" where the "userid" is taken from the most recent $USERID card in the job. The file is accessed for reading under the file code "g*". Since no LUD is specified, the batch processor will assume that the file will only be used in this job step

$ prmfl s*/x1s,r,s,alibaba/fort$cave

This specifies a sequential permanent file named "alibaba/fort" with a file password of "cave". The file is being accessed for reading only. The program for this activity will refer to the file using the file code "s*" and the JCL will remember this file by the LUD "x1". The disposition code "s" indicates that the file is to be saved for future activities in this job.

$ prmfl **/t3,e,r,/cat/dogprog

This accesses the random file "userid/cat/dogprog" with execute permissions. The file code is "**" and the LUD is "t3". A disposition code of "r" is assumed, so the file will be released at the end of this job step.

Before execution of a batch job begins, the Peripheral Allocator (PALC) will check to make sure that files referenced in $PRMFL cards actually exist. It will also make sure that the userid specified on the most recent $USERID card has the necessary permissions to access the given file with the requested permissions. If the file does not exist or the user does not have the right permissions, the job will not be executed at all.

4.5 The $FILE Card

The $FILE card has two major uses: it can be used to create a temporary file, and it can be used to refer to a file that has been passed down from a previous job step. (Files are passed down by specifying a disposition code of "s".)

The general format of the $FILE card is

```
$      file    fcode,lud,access,options
```

"fcode" is a file code for the file. "lud" is a logical unit designator; this can include an optional disposition code. "access" consists of a number followed by a letter. The number gives the size of the file in *links*. A link is 12 *llinks* long, or 3840 words. The letter is either "r" indicating that the file is random or "l" indicating that the file is sequential or "linked". Thus an "access" of "10l" creates a sequential file that is 10 links (120 llinks) long. The number of links may be omitted in this field if the file already exists (i.e. it has been passed down from a previous step). If the file does not already exist and no number is specified, the file is created with a size of one link (12 llinks).

The possible "options" which can be specified on the $FILE card are described in the GCOS8 JCL Reference Manual.

Below we give some examples of ways in which the $FILE card can be used.

$ file p*,d1s,8l

This references a sequential file with file code "p*" and LUD "d1". If a file with LUD "d1" does not already exist, a temporary file is created with a size of 8 links (96 llinks). The disposition code "s" tells the batch processor to save the file for future activities at the end of this job step.

$ file h*,k8,r

This references a random file with file code "h*" and LUD "k8". If a file with LUD "k8" does not already exist, a temporary file is created with a default size of one link (12 llinks). Since no disposition code is specified, "r" is assumed and the file is released at the end of the current activity.

There is one more way that the $FILE can be used. Occasionally, a program will create output that you aren't interested in keeping. In this case, you might want to create a "bit bucket", i.e. a "junk" file which will receive the unwanted output and then get rid of it. To do this, you specify a card of the form

```
$       file    fcode,NULL
```

where "fcode" is the file code which will receive the unwanted output. For example, if a program writes unwanted output to the file code "ot", the card

```
$       file    ot,null
```

will create a bit bucket for that output. Note that this should only be used for *output* files.

4.6 The $ DATA Card

The $DATA card is used when you are including data in the same input stream as your JCL (as opposed to obtaining the data from a permanent or temporary file). The $DATA card has the format

```
$       data    fcode,options
```

where "fcode" is the file code to be associated with the input data. The lines of data immediately follow the $DATA card in the input stream, as in

```
$       program tf
$       data    cz
tf /stuff.t >/outstuff index=/outindex
```

The TF text formatter expects its command line to be given under the file code "cz". Rather than storing the command line in a permanent file, you can just include it in the input stream as shown above.

Normally, the batch processor will continue to take in input data after a $DATA until it encounters a line with a "$" character in column one. This line is taken to be a JCL card, and JCL scanning begins once more. If some of your input data lines have the "$" sign in column one, you can begin the data with a card like

```
$       data    cz,copy
```

When the COPY option is specified on the $DATA card, the batch processor will take in input data after the $DATA until it encounters a card of the form

```
$       endcopy
```

Any JCL cards which occur between the $DATA and $ENDCOPY cards are simply taken as lines o

Any JCL cards which occur between the $DATA and $ENDCOPY cards are simply taken as lines of data for the data file (with one or two exceptions outlined in the GCOS8 JCL Reference Manual). Thus

```
$       data    gh,copy
$       fortran
$       endcopy
```

stores a $FORTRAN card under the file code "gh".

For some activities, the $DATA card is not required when data is included in the job input stream. For example, you can include your source program immediately after the $FORTRAN card in a Fortran compilation activity. The batch processor will automatically assume that there is a $DATA S* card immediately preceding the Fortran source code. (If you want, you can put a $LIMITS card between the $FORTRAN card and the input data.) You can also omit the $DATA for PL/I compiling, GMAP assembling, and certain other activity types which are recognized by the batch processor.

There are times when you might want the input data from one job activity to be saved for subsequent activities. To do this you follow the file code on the $DATA statement with the letter "s" as in

```
$       data    c4s
```

This has the effect of storing the input data in a temporary file which has an LUD that is the same as the file code. Thus the above card would store the data in a temporary file with LUD "c4". The input data could be used in later activities by referencing the file with the LUD "c4".

There is one more useful feature of the $DATA card. Sometimes you would like to associate a null input file with a particular file code. Thus the first time this file is read, the program will receive an indication of "end-of-file". To do this you include a card of the form

```
$       data    fcode,NULL
```

in your JCL. This will create a null input file with the file code "fcode". (Note that this is similar to using the NULL option on the $FILE card when you want to specify a junk *output* file.)

4.7 SYSOUT

The SYSOUT queue contains output which is waiting to be printed on the high-speed printer. By default, anything written to the file code "p*" is sent to the SYSOUT queue. Thus a program can expect that anything written to file code "p*" will go to the printer. (Of course, you can redirect "p*" with the right JCL statements; for example,

```
$       prmfl   p*,w,s,/out
```

will redirect output from the printer to the permanent sequential file "userid/out".)

The $SYSOUT card can be used to associate other file codes with the SYSOUT queue. The usual

format of the card is

```
$      sysout  fcode,ORG
```

where "fcode" is the file code to be associated with the SYSOUT queue. The ORG on the end indicates that your output is to be associated with the printer at the job's point of origin. Thus if the card

```
$      sysout  ot,org
```

appeared in the JCL for an activity, anything which that activity's program sent to file code "ot" would end up on the printer.

Below we give a sample program using the file JCL cards we have discussed in this section.

```
$      ident   ppan$wendy,tinkerbell
$      option  fortran
$      fortran
$      prmfl   s*,r,s,/fortsrc
$      gmap
$      prmfl   g*,r,s,/gmapsrc
$      prmfl   p*/e1s,w,s,/listout
$      execute
$      limits  10,20k,,4k
$      file    05,e1,l
$      file    19,,4l
$      data    41
Far and few, far and few,
Are the lands where the Jumblies live...
$      sysout  10,org
```

The above program first compiles the Fortran source which is found in the sequential permanent file "userid/fortsrc". It next assembles the GMAP source found in the sequential permanent file "userid/gmapsrc". Note that the program redirects "p*" in this case, so that the output listing from the GMAP compiler goes to the sequential permanent file "userid/listout". Furthermore, the disposition code "s" saves this file for use in the next activity of the job. The final activity loads the compiled Fortran and GMAP modules together and executes them. The program accesses "userid/listout" under the file code "05", it accesses a nameless four-link temporary file under the file code "19", and it accesses the two lines of input data under the file code "41". Finally, it creates output under the file code "10" which is sent to the printer.

5. The Loader

The loader takes the object decks which compilers and assemblers generate, and creates core image files which can actually be executed by the system. When the loader is invoked by a $EXECUTE card in a batch job, it will create the required core image file and then execute it.

The loader expects the object decks which it works with to be stored under the file code "b*". Wher a compiler or assembler creates an object deck during compilation, it stores the deck under this file code automatically and arranges that the file code is passed on to subsequent activities. If there are several compilation/assembly activities in a job, each activity adds its object decks to the "b*" which has been accumulated in the previous job steps. When the $EXECUTE activity begins therefore, the loader obtains the object decks that were produced during all the previous compilations and assemblies. (At the end of the $EXECUTE activity, the loader will release the "b*" file; thus subsequent compilations or assemblies in the job will be working with a brand new "b*".)

In addition to the "b*" file, the loader makes use of a file with the file code "r*". This file contains loader directives which govern the way in which the loader prepares and executes the object decks it receives.

Loader directives find their way to "r*" in a somewhat unusual way. The loader directives are actually placed in the JCL input stream as special JCL cards. For example, the $OPTION card which we have shown in our job examples is actually a loader directive. As the batch processor reads the JCL for the job, it collects the loader directives which it finds in the JCL stream and stores them in "r*". By the time the job reaches the $EXECUTE activity, the "r*" file ought to contain all the directives the loader needs. In the sections to come, we will examine the most commonly used loader directive JCL cards.

Before we move on to these loader directives though, we will mention another important file code used by the loader. When the loader has linked all the necessary object decks together and has loaded the result into memory, the core image of the program can be copied out onto a file and saved. Once you have saved this core image file of your program, you can execute your program from this file rather than having to recompile and reload. The loader will save this core image under the file code "h*" (see the description of the SAVE option in Section 5.1).

Note that you do not have to specify $FILE or $PRMFL cards for any of the files "b*", "r*", or "h*". This is all handled automatically for you by the system.

5.1 The $OPTION Card

The standard format for the $OPTION card is simply

```
$      option  options
```

There are quite a few "options" that can be specified on this card, but we will only list the most commonly used. For full details, see the GCOS8 JCL Reference Manual.

The FORTRAN option is required when loading programs generated by the Fortran compiler. This is actually a compound option that sets up all the loader directives that are needed to handle Fortran programs. There are similar options named FORT77, COBOL, CBL74, and PLI. For loading Pascal and B programs, the NOFCB option is used (see Section 6.3); Pascal is not standard GCOS8 software and hence does not have an option name all to itself.

software and hence does not have an option name all to itself.

The MAP option instructs the loader to generate a detailed memory map on the output listing for the job. This shows where each routine was loaded into memory, and can be useful when examining dumps. The NOMAP option tells the loader not to generate a memory map. If neither option is specified, the default is MAP.

There are three options which determine whether the loader will attempt to execute the program once the core image file "h*" has been produced. The NOGO option indicates that execution should not take place once the program has been loaded. The GO option indicates that the program should be executed provided no fatal or non-fatal errors were detected during loading. The CONGO (CONditional GO) option indicates that the program should be executed unless some fatal errors were detected during loading. The default is CONGO.

One reason you might want to use the NOGO option would be if you were just going to take the "h*" file that the loader produces and save it in a permanent file. Once this is done, you can execute that file directly with a $PROGRAM card, without putting the program through the loader again. To do this, you use the "SAVE/name" option, where "name" is a maximum of six characters. You must also provide a $PRMFL card with a file code of "h*" to tell the system where to store the core image file. This is done in the following two job steps.

```
$       option  save/prog1,nogo
$       gmap
$       prmfl   s*,r,s,/gmapsrc
$       execute
$       prmfl   h*,w,r,/cat/myprog
```

The "name" specified in the SAVE option is "prog1". When the loader has prepared the core image file, it will send it to the specified file code "h*". Thus the core image file will be stored in the permanent random file "userid/cat/myprog", and it will be stored under the name "prog1". To execute this program in future, you would begin with a card of the form

```
$       program prog1
```

We will say more about this in Section 6.

Options can be specified on multiple $OPTION cards or on the same card, separated by commas. For example,

```
$       option  fortran,symref,nogo
```

is equivalent to

```
$       option  fortran
$       option  symref
$       option  nogo
```

Since $OPTION cards are just copied into the "r*" file for later use by the loader, they can be placed just about anywhere in the job before the $EXECUTE activity that uses the loader directives

Generally speaking, good programming style dictates that you should try to associate options with the activity which generates the object decks involved. Thus in our examples we placed the FORTRAN $OPTION card immediately before the $FORTRAN card that compiled the program to be loaded.

5.2 The $LOWLOAD Card

For historical reasons, the loader is inclined to load programs into the high addresses of memory rather than the low addresses. The situation is this: your program is allocated a certain chunk of memory. If left to its own devices, the loader would put the first routine of your program into the top of that chunk, and stack subsequent routines underneath the first one. Thus your program would be loaded from high addresses down towards zero.

Needless to say, this is a less than logical situation nowadays. A card of the form

```
$       lowload
```

will cause the program to be loaded in the more natural manner from low addresses up. You .ul must specify $LOWLOAD for programs written in B, Pascal, Algol, and PL/I.

The $LOWLOAD card must be the first thing that is put into the "r*" file of loader directives. Thus the $LOWLOAD card should be included in your job before any of the job's activity cards.

5.3 The $LIBRARY Card

To specify libraries of object decks which can be used during the loading of a program, you enter a card of the form

```
$       library fcode1,fcode2,...
```

where "fcode1,fcode2,..." are all file codes. This card may appear anywhere in the program before the $EXECUTE step. In the JCL for the $EXECUTE activity, you must include cards that identify the file codes which are listed on the $LIBRARY card. For example, consider the pair of job steps below.

```
$       lowload
$       option  fortran
$       fortran
$       prmfl   s*,r,s,/fortsrc
$       library x1,x2
$       execute
$       prmfl   x1,r/c,r,/lib1
$       prmfl   x2,r/c,r,/lib2
```

The $LIBRARY card names two file codes, "x1" and "x2". Thus the $EXECUTE activity has to contain JCL cards identifying those two file codes. The two $PRMFL cards serve this purpose. Note that the libraries are accessed with "read concurrent" permission so that other users are not

prevented from using the libraries at the same time.

The loader uses the specified libraries to resolve external references which occur in the program being loaded. For example, suppose the Fortran program being loaded above contained an unresolved reference to a function called "func". The loader would search the two libraries for an occurrence of the function "func"; if one was found, it would be linked in with the Fortran program being loaded.

Libraries are searched in the order in which they appear on the $LIBRARY card. In the example above, file code "x1" (/lib1) would be searched before file code "x2" (/lib2). Thus if both "/lib1" and "/lib2" contained a function named "func", the loader would find the one in "/lib1" first and that would be the function linked in with the program.

If a reference is still not resolved after the libraries on the $LIBRARY card have been searched, the loader will search file code "*L" and then "L*". These are standard system subroutine libraries. For example, the Fortran SIN and COS routines are found on these libraries.

5.4 The $USE Card

The $USE JCL card can be used to force the loader to load one or more routines from a library. The general format of the card is

```
$      use     rout1,rout2,...
```

where "rout1,rout2,..." are the names of the routines to be loaded.

The $USE card has a similar purpose to the "use=" options of some TSS compilers. In B, for example, routines which handle floating point numbers are not usually loaded with a program unless the program explicitly uses such numbers. However, you might want to be able to read or write floating point numbers without actually doing calculations with them, in which case you would have to obtain the floating point routines somehow. In TSS, you would compile your B program with the "use=.float" option. In batch, you would specify the loader directive

```
$      use     .float
```

The $USE card can appear anywhere in the job before the $EXECUTE card which starts the loading process.

5.5 The $FFILE Card

The $FFILE card refers to files with non-standard or "funny" formats. This card has a large number of arguments and specialized uses that we won't go into here. The only use we will describe occurs when you wish to associate a number of Fortran I/O units with a particular file.

Suppose you want to associate the Fortran output unit numbers "06" and "19" with the printer. You could do this with a card of the form

```
$       ffile   p*,lgu/(06,19)
```

The "lgu" stands for "logical unit". When the loader sees a $FFILE directive of the above form, it will associate the given output units with the file code "p*" so that whenever the Fortran program writes to units "06" or "19", the output is actually sent to file code "p*" instead of file codes 06 and 19. Remember that the $FFILE card is a *loader* directive; it tells the loader something about a particular file code, but it does *not* associate an actual file with that file code. To access an actual file you use $FILE, $PRMFL, or $SYSOUT (when you want the file to go to the printer).

Note that the FORTRAN COMPILER automatically associates Fortran units 06 and 42 with the printer, units 05 and 41 with the file code "i*" and units 07 and 43 with the card punch. Thus you do not need $FFILE cards if you are just using these default units.

Unlike many other loader directives, the $FFILE card should appear in the body of the loading activity, after the $EXECUTE card.

6. Executing Previously Compiled Programs

If your program is compiled or assembled in one activity, we have already shown how it can be executed in a later activity by calling the loader with $EXECUTE. Naturally though, there are numerous occasions when you will want to run a program without having to compile it first. In this section, we will talk about ways to do this with the $PROGRAM card. This card can be used for programs written in most of the programming languages supported on this system. Unfortunately, programs written in Fortran and Algol are special cases which must be executed in a different way. This different way is described in Section 6.1.

In Section 3.2 we said a little about the $PROGRAM card. This card is used to specify the start of an activity and to name what program will be executed by that activity. However, the $PROGRAM does not tell the system where to find the core image file where the specified program is stored. Thus there must be some other way to tell the system how to get the program you want it to run.

This is done with a $PRMFL card which has the special file code "**". For example,

```
$       program tf
$       prmfl   **,q,r,tf/1.0/mod/tf.h
```

tells the system that you are going to execute a program called "tf" that is located in the random permanent file called "tf/1.0/mod/tf.h". (Note that we use "query" permission so that the TF "h*" file can be shared with other users.) This information is all the system needs to be able to find the program you want. Of course, your JCL for the activity must also include definitions for the file codes which the given program uses; in the case of TF, you would at least need a file with file code "cz" to pass TF its command line (as mentioned in Section 4.6).

The file referenced with file code "**" must contain core images which have already been prepared by the loader. In Section 5.1 we showed how the SAVE option of the $OPTION card could be used to save the "h*" file prepared by the loader. The "h*" file was saved under a name of no more than

six characters in a permanent file. We gave the example of saving an "h*" under the name of "prog1" in the file "userid/cat/myprog". If we wished to execute that program later, we would use the pair of JCL cards

```
$       program prog1
$       prmfl   **,e,r,/cat/myprog
```

The batch processor would obtain the program "prog1" from the given file and then execute it.

The random permanent files which contain "h*" core images are sometimes called "qstar" files. This is because the special system file code "q*" is often used for files which contain system-loadable elements. A qstar file can contain more than one "h*" type element. You can keep all your program core images in one big qstar file, if you like; when you want to execute one of the core images, all you have to do is specify the right name and the batch processor will pluck out the right element. More often though, people store each of their "h*" core images in separate files.

The L system command will list the "h*" elements of a qstar file in an easy-to-read format.

6.1 RLHS

RLHS is a system program which will execute "h*" files which originally came from Fortran, Algol, or Cobol source. The reasons why such programs need special handling are too complicated to go into here; for a full explanation, you'll have to go to the GCOS8 software reference manuals.

Suppose you have compiled a Fortran program, prepared a core image file with the loader, and stored that "h*" in a qstar file named "userid/qst". The cards

```
$       program rlhs
$       prmfl   h*,e,r,/qst
```

will obtain the "h*" file from "/qst", will load it, and will execute it. If the program was loaded with a "SAVE/name" option on the option card, you use a card of the form

```
$       program rlhs,,=name
```

This will cause the "h*" file named "name" to be plucked out of the qstar file and executed.

Below we give an example of three job steps which compile a Fortran program and execute it twice with two different input files.

```
$       option  fortran,save/lamp
$       fortran
$       prmfl   s*,r,s,/lampsrc
$       execute
$       ffile   p*,lgu/(06,19,42)
$       prmfl   h*,w,r,/lamph
$       prmfl   05,r,s,/lampin1
$       program rlhs,,=lamp
$       prmfl   h*,r,r,/lamph
```

```
$       prmfl   05,r,s,/lampin2
```

The Fortran program has one input unit (05) which is associated with "userid/lampin1" the first time the program is executed and with "userid/lampin2" the second time it is executed. The program has three output units (06,19,42) which are all associated with the printer.

6.2 The $SELECT Card

Many system programs like TF do not have their own activity cards and so must be executed with a $PROGRAM card. If you use such programs frequently, it's nice to be able to avoid typing in the same four or five JCL cards over and over again. The $SELECT card allows you to do just that.

The card has the form

```
$       select  pathname,subs
```

Whenever the batch processor encounters a $SELECT card in the JCL stream of a job, it replaces the card with the contents of the file "pathname". (This process works in much the same way as the ".so" command does in TF.) Thus you can store commonly used JCL cards in permanent files and use the $SELECT card to copy in the JCL whenever you want to use it.

There are a number of useful "select files" stored under the catalog "cc" at Waterloo. For example, the file "cc/tf" contains

```
$       program tf
$       prmfl   **,q,r,tf/1.0/mod/tf.h
$       limits  4,24k
$       data    cz
```

The file "tf/1.0/mod/tf.h" contains the TF hstar; the "q" permission on the $PRMFL card is much like "read" or "execute" permission except that it allows more than one person to read the file at a time. Using the above file, you could TF something in batch with the JCL

```
$       select  cc/tf
tf /infile >outfile
```

The $SELECT card is replaced by the contents of "cc/tf", i.e. the JCL cards we have just shown. Thus the TF command line comes immediately after the $DATA card; the input data will therefore be associated with the file code "cz", just the way you would want.

The $SELECT card is capable of more than just copying the contents of a file into the input stream; it will make substitutions as well. You can specify constructs of the form

```
keyword=(string)
```

as substitution arguments for $SELECT. Then as the $SELECT copies the contents of the given file into the input stream, it will replace all occurrences of the construct "&keyword" with the given "string". As an example of this, suppose "userid/fort" contains

```
$       option  fortran
$       fortran
$       prmfl   s*,r,s,/ftn/&src
$       execute
```

The card

```
$       select  /fort,src=(xprog)
```

will be replaced by

```
$       option  fortran
$       fortran
$       prmfl   s*,r,s,/ftn/xprog
$       execute
```

The string "xprog" is inserted in the place of "&src". In this way, you can use the one $SELECT card to compile and execute any Fortran source file in the catalog "userid/ftn". JCL for any files used by the Fortran program can be listed after the $SELECT card; this will have the effect of listing the file JCL cards after the $EXECUTE card that is in "/fort".

You should bear in mind that any file that is being selected with the $SELECT card should be in BCD format; after all, $SELECT just copies the contents of the "select file" into the JCL input stream and this stream has to be in BCD. If you want to include an ASCII file in your JCL stream, you should use the $$SELECT option of the JRN command (see Section 8.3). This works in much the same way as $SELECT, but JRN will convert the selected file to BCD as it is passed on to the batch processor.

The proper use of $SELECT cards can greatly reduce the amount of typing you have to do to prepare a job's JCL. Making use of the "select files" that are already stored under the "cc" catalog is also helpful. For example,

```
$       select  cc/ranedit
```

provides the JCL needed to use the RANEDIT program to update your libraries of compiled object decks. RANEDIT picks up the "b*" object decks produced by compilers and assemblers and then puts those compiled routines into a user library. If there are routines of the same name already in the library, they are replaced by the newly compiled routines.

RANEDIT works in a kind of sneaky way: it masquerades as the loader. After all, RANEDIT has many things in common with the loader: it takes "b*" object decks as input, and it can make use of many of the loader directives which are passed along in the "r*" file. Thus "cc/ranedit" contains the JCL

```
$       execute
$       limits  01,26k,,2000
$       prmfl   **,q,r,cmdlib/ranedith
$       sysout  43,org
```

```
$       file    sc,z10r,20r
```

In the same way that specifying a "**" file allows the batch process to find a program for $PROGRAM, specifying a "**" in a $EXECUTE activity tells the batch processor to make use of something in place of the loader. The other cards of the file specify file codes which RANEDIT needs to operate.

6.3 Compiling and Running Pascal

As an example of the work we have been doing in the past few sections, we will now describe how to compile and run a program in Pascal. This is a little different from running programs written in other languages, because Pascal is not standard GCOS8 software. Thus there is no such thing as a $PASCAL card, for example. There are however a number of "select files" which solve a lot of the Pascal programmer's problems.

The minimum JCL to compile and execute a Pascal program is given below.

```
$       ident   user$pswd,banner
$       lowload
$       option  nofcb
$       program pascal,lstin,debug
$       select  pascal/compile
$       prmfl   s*,r,s,/source
$       select  pascal/execute
$       prmfl   i*,r,,/data
```

The card selecting "pascal/compile" must follow the $PROGRAM card immediately. The options "lstin" and "debug" on the $PROGRAM card instruct the Pascal compiler to produce a source listing and to include range-checking code in the compiled program as an aid to debugging. A complete list of options which can be specified on the $PROGRAM card is given in "expl pascal batch options". The $PRMFL card with file code "s*" may be replaced by Pascal source code placed right into the JCL input stream.

The second $SELECT card is replaced by code that loads the compiled program and executes it with a $EXECUTE card. Three libraries are automatically specified for the loading process. These have file codes "pl", "bl", and "bb". If you want to include your own program libraries when the program is being linked, you must specify your own $LIBRARY card with a different file code. For example,

```
$       library me
$       select  pascal/execute
$       prmfl   me,r,r,/mylib
```

adds the library "userid/mylib" to the standard system libraries.

If your Pascal program needs data, it is supplied under the file code "i*" as shown. Naturally, this data can also be placed in the JCL input stream under this file code.

To save a compiled program for later execution, you can simply use the SAVE option. Thus

```
$       option  nofcb,save/frodo
$       select  pascal/execute
$       prmfl   h*,w,r,/hobbiton
```

will save your program under the name "frodo" in "userid/hobbiton". To execute this program now, all you have to say is

```
$       program frodo
$       prmfl   **,e,r,/hobbiton
```

For compatibility with TSS, a batch program will accept a TSS-type command line under the file code "cz". Thus if the "frodo" program needed a command line, you could say

```
$       program frodo
$       prmfl   **,e,r,/hobbiton
$       data    cz
x <userid/infile  >>userid/outfile
```

The line of data in "cz" is passed to the Pascal program as a command line.

We have mentioned that Pascal programs look for their input under the file code "i*" by default. Your program can work with other file codes by using a simple naming convention. The Pascal statement

```
openf( f, 'fc*ot', 'w');
```

will associate the file variable "f" with the file code "ot". If a $FILE or $PRMFL card associates the file code "ot" with a particular file, that will be where the output from the Pascal program will go. If no such card exists, the batch processor will create a temporary file with file code "ot". Naturally, such temporary files are only created for output files; you must have a $FILE or $PRMFL card for files which the program intends to read.

For further information about using Pascal in batch, see "expl_pascal_batch".

7. GCOS Batch Flow

In this section, we'll discuss the various stages a batch job goes through from the time it is invoked to the time it terminates. This will help you to understand what's going on when you run a batch job using the monitoring facilities of TSS. These TSS/Batch interface routines will be described in Section 8.

7.1 GEIN

The system program which actually reads the JCL for your job is called GEIN. When GEIN is simply reading in your JCL the TSS/Batch interface routines will report "reading-rmt". Next, GEIN

performs simple syntax-checking and replaces $SELECT cards with the contents of their "select files". GEIN also checks that the password on the $USERID card matches the password of the specified user, and makes several other minor checks. During this error-checking stage, the TSS/Batch interface routines will report "executing".

If GEIN detects an error in your JCL, the job never gets any further; it is simply terminated and an output listing is issued with appropriate error diagnostics. If GEIN accepts your job, the JCL is stored in a special area and the system scheduler is notified.

7.2 The System Scheduler

When GEIN is finished, the system scheduler (RGIN) takes over. RGIN examines the stored JCL and decides which job queue the job should be placed into. This decision is based upon the total amount of processor time and the maximum amount of memory which the job might need.

Some job queues are not operative all the time. For example, jobs which use magnetic tapes are placed in the ".tape" queue. This queue is only open when there is an operator on duty (since at other times of day, there won't be anyone around to mount your tape). If your job is put into an unopened queue, you should ask the operators to open the queue (if there are any operators on duty). They will do this for you when they decide that your job can be run without seriously impacting other users.

You can see how many jobs are waiting in the batch queues and how many jobs are executing, with the TSS system command

```
queue -all
```

For further information about this command, see "expl queue".

Once the job is in an opened queue, it will wait until all the jobs ahead of it in the queue have passed on to the next stage of the batch process flow.

7.3 PALC

When a job reaches the front of its queue and the system decides it can be executed, the job is passed on to the Peripheral Allocator or PALC. PALC is the system program which actually decodes your JCL and gets things ready for the first activity of your program to run.

There are several messages which the TSS/Batch interface routines may report while the job is being processed by PALC.

wait-aloc
: This indicates that the job is waiting for PALC to look at it. This occurs if PALC happens to be swapped out of memory or if PALC is working on another job.

wait-perip
: This indicates that PALC is trying to allocate peripheral devices and files for the next activity. For example, if your program requires a tape to be mounted, PALC will stay in the "wait-perip"

For example, if your program requires a tape to be mounted, PALC will stay in the "wait-periph state until there is a free tape drive available. PALC may also enter this state if you have asked to use a disk file which is busy at the moment.

in limbo

This indicates that PALC has asked the operator to obtain a tape or disk from the site library. Your job will stay in this state until the operator indicates that he or she has found the tape and has it ready on hand.

wait-media

Once the operator has obtained any tapes or disks your program needs, PALC will tell the operator where the media should be mounted. Between the time that the "mount" message is issued and the time that the operator indicates that the medium has been mounted, your job is in the "wait-media" state.

When PALC is notified that your first activity's peripheral devices are all ready, it prepares your activity for memory allocation. It does this in a rather sneaky way. Activities that are actually running are occasionally swapped out of memory to make room for higher priority jobs; activities are also swapped out when they are waiting for one thing or another (e.g. when they are waiting to be allocated more storage if they are growing). PALC doctors up your first activity and puts it into the same format as a job that has just been swapped out of memory. It then waits for a chance to pass the activity on to the core allocator (CALC). During this process, the TSS/Batch interface routines report "wait core".

7.4 CALC

CALC will be passed the first activity of your job as soon as PALC has prepared it in "swapped-out" form. To CALC, it is just one more job that has been swapped out of memory and it will be swapped in as soon as memory becomes available. Thus there is really no difference between the first time the activity is swapped into memory and any subsequent time.

While an activity is being executed, the TSS/Batch interface routines will intermittently report two different messages: "xx-executing" or "xx-swapped" where "xx" is a number which indicates which activity of your job is currently working. For example, "01-executing" indicates that the first activity of your job is currently in memory and working. (Thus the presence of the activity number distinguishes this stage from the system scheduler stage.) "01-swapped" means that your first activity has been swapped out of memory for the moment, but will go back into memory as soon as the opportunity presents itself.

7.5 Termination

When an activity terminates, the TSS/Batch interface routines will report "xx-terminating" where "xx" is the number of the activity. Accounting takes place at the end of every activity; thus you are charged by the activity, not by the job. Once the accounting is finished, the job will go back to PALC. If the job has more activities to be executed, PALC will start all over again, re-allocating peripherals and so on. Thus your job will loop through PALC, CALC, and the termination

accounting step for each activity in the job.

8. TSS/Batch Interface Routines

In this section, we will look at the ways TSS allows you to invoke batch programs, to monitor their progress, and to examine their output.

8.1 Submitting Batch Programs

The easiest way to prepare a batch program is to use a TSS text editor like FRED. Once you have done this, you store the JCL for the program in a file so that you can pass the file on to the batch processor. This file can be either temporary or permanent; however, this is the only time a temporary file can be passed to a batch job. Temporary files *cannot* be directly passed to batch jobs as data files; however, you can pass the contents of a temporary file by using the $$SELECT option of the JRN command (see Section 8.3).

The TSS command which submits this file of JCL to the batch job flow is JRN. Thus the system command

```
jrn /myfile
```

submits the contents of "userid/myfile" to the batch processor. The system will reply with the snumb it has generated for the submitted job, and then you will be ready for more TSS commands.

One convenient feature of JRN is that it prepares a $USERID card for you. Thus you can store your JCL in permanent files without having to include your password. Not only is this more secure (since your password is no longer sitting out on disk where prying eyes might find it), but it avoids the problem of having to change your JCL files every time you change your password. You can also allow other people to copy your JCL files without worrying about them running their jobs under your name.

The simple JRN command above just submits your batch job and then lets you go on to other things. Often though, you would like to be able to monitor your batch job to see what it's doing. To do this, you would say

```
jrn /myfile:moni
```

This command will print information to your terminal as the batch job progresses. It will print a line every time your job passes from one stage in normal batch flow to the next stage. During long stages (e.g. execution) it will print a line every five seconds or so to indicate that the job is still running. The output lines will contain the snumb of the job that is running and the report messages described in Section 7. Below we give a sample of the monitor output from a two-activity batch run

```
*jrn junk:moni
snumb k848t
k848t -01   wait-aloc    @ 10.850
k848t -01   wait-core    @ 10.850
```

```
k848t -01   wait core     @ 10.850
k848t -01   executing     @ 10.851
k848t -01   terminating   @ 10.852
k848t -02   wait-perip    @ 10.852
k848t -02   executing     @ 10.853
k848t -02   terminating   @ 10.857
k848t output waiting      @ 10.857
normal termination
```

Of course, there is always the chance that your job will have to wait for a long time in a particular step; for example, it might get into a long queue awaiting execution, or it might wait a long time for a tape to be mounted. If you would rather break off your monitor output partway through and move on to other TSS commands, just hit "break".

If you have a job that is running in the system, you may want to look in on it from time to time to see how it is coming along. To do this, you can use the TSS JMON command. The command

```
jmon snumb
```

will locate the batch job with the given snumb and will begin issuing monitor output lines, just like the monitor output from JRN. JMON is clever enough to find your job if it is sitting in one of the job queues waiting to be allocated; however, it has no way of finding anything out about a job after the job has terminated. In this case, it will simply give the message "job not found". To break out of JMON, simply hit "break" or "attn".

If your job finishes while being monitored by JMON, JMON will automatically call JOUT. JOUT is explained in the next section.

8.2 JOUT

When your job has finished, you will probably want to examine its output. Output to a permanent file can of course be examined just by listing the file. Output which has been sent to SYSOUT is harder to get at. Many times you don't want to wait for the output to be printed and for the listing to be put out in the output boxes. In this case, you can use the JOUT subsystem to examine your output.

JOUT is able to get at any output which has been sent to SYSOUT but which has yet to be printed. You can get into JOUT in several ways. First of all, you can specify it on the JRN command line.

```
jrn /jclfile:jout,moni
```

will run the job in "userid/jclfile", monitor it, and then call JOUT to examine the SYSOUT output. (If the "moni" option is not specified, your job will be submitted and you will return to TSS system level; however, the output from your job will be held and will not be sent to the printer until you have examined it using JOUT.) Another way to get into JOUT is to use JMON to monitor your job; as we mentioned in the last section, JMON automatically calls JOUT when the job terminates. Lastly, you can call JOUT directly from the terminal.

```
jout snumb
```

will allow you to examine the SYSOUT output from the job with the given snumb, provided the output hasn't been printed yet.

Regardless of how you enter JOUT, you will always be asked the question

```
function?
```

JOUT wants to know what you want to look at first. The possible answers to the "function?" question are listed below.

activity n
: This tells JOUT that you want to look at the output from the n'th activity of your job, where "n" is some appropriate integer. If you don't specify an "n", it assumes you want to look at the first activity.

list
: This lists all the report codes associated with the activity you're currently looking at. Report codes are explained in detail in Section 9.4; for the moment, you can think of report codes as two-character titles for individual output listings generated by your program. If you are looking at the output from a Fortran program, the report codes are just the Fortran unit numbers in octal For example, output unit 42 has report code 52. System software programs often send their output listings to report code 74; e.g. this is where the Fortran compiler sends its compiler listing and where the loader sends its memory map.

  The LIST function will always show a report named "$$". This report contains a listing of the job's JCL and other job statistics.

print rep-code
: This prints the contents of the given report code. For example, "print $$" will print your JCL listing. The "print" function realizes that the SYSOUT listing was sent to the printer, while JOUT is likely only working at a terminal; thus "print" tries to shorten the printer-length lines by suppressing multiple blanks. In other words, "print" converts all occurrences of multiple blanks into single blanks.

eprint rep-code
: This is the same as "print" except that it does not suppress multiple blanks. You will be able to see exactly how things will come out on the printer. (Of course, this may lead to a lot of lines wrapping around on the terminal, especially if you are at a CRT with a screen that is only 80 characters wide.)

copy rep-code;filename
: This will copy the contents of the given report code into a file with the specified filename. This is handy if you want to edit the output with a text editor; it is also handy if you just want to keep the output around on the system for a while.

call TSS-command
: This will invoke the given TSS command. For example, "call expl jout" will explain JOUT and

its options if you happen to forget one of the option formats. Important note: do not call LSTQ, BW, or BKDR from JOUT, or you'll get blown out of the water.

width n

This sets the width of the lines which JOUT prints out. The default width is 128 characters for 2741 terminals, and 72 characters for other terminals. Output lines from JOUT will be broken and folded at the specified column.

scan rep-code

This will allow you to scan through a given report code. We will describe how to use this option in Section 8.4.

In addition to the JOUT functions listed above, there are three "exit" functions. These three all cause you to leave JOUT and return to system level. They also tell JOUT what to do with the output from the batch job.

direct onl

This tells JOUT to send your job's output on to the printer. Thus you will eventually receive a print listing of every report code. Note that if you misspell the "onl", your output will be directed to some fictitious print queue and you'll have to ask the operator to get it back for you. (There's a chance that the operator won't be able to get at your output either; in this case, you'll have to re-enter JOUT and type in the "direct" command correctly.)

release

This releases all the report codes of your job. Nothing will go to the printer.

hold

This puts your printer output on "hold". In other words, it is kept in the system but nothing is done with it. You will have to enter JOUT again to specify whether the output should be printed or released.

You must specify the ORG option on $SYSOUT cards if you wish to examine the associated file codes using JOUT; otherwise, JOUT will not be able to get hold of the output.

8.3 $$Cards

$$ cards are special cards that can occur in files which are executed by JRN. These can be regarded as control cards for JRN. For example, the card

```
$$jout,moni
```

at the beginning of a file will specify the "jout" and "moni" options for JRN, even if the options aren't specified on the JRN command itself.

Another commonly used $$ card is $$SELECT. This serves much the same purpose as the $SELECT JCL card. For example, if a file contained

```
$       ident   userid$password,banner
$       option  fortran
```

```
$      fortran
$$select(/fortsrc)
$      execute
```

JRN would replace the $$SELECT card with the contents of the file "userid/fortsrc" and then submit the result to the batch processor. Note that the files included in the JCL stream with $$SELECT may be either temporary or permanent.

All $$SELECT cards are handled by the JRN command; $SELECT JCL cards are expanded by GEIN during normal batch job flow. The difference is that $SELECT copies in the file directly, and therefore the file must be in BCD format. The $$SELECT card is part of JRN and therefore can convert ASCII files to BCD. Thus when you are selecting an ASCII file, you must use the $$SELECT card of JRN; if you are selecting a BCD file, you can use either $$SELECT or $SELECT.

For more about $$ cards, see the GCOS8 TSS Terminal/Batch Interface Manual.

8.4 SCAN

The SCAN function in JOUT can be used to locate lines of output in a given report code without having to read through the whole listing.

You begin by replying "scan rep-code" to JOUT's "function?", where "rep-code" is the report code you want to look at. The system will then ask "form?". SCAN is asking for the format of the listing you are examining. SCAN recognizes several different standard listing formats, and specifying the format of the listing will help the subsystem look for the things you want to find. The possible forms are

gmap
: With this form, the subsystem will begin by telling you the number of assembly errors specified in a GMAP listing.

fort
: With this form, the subsystem will begin by telling you the number of compilation errors specified in a Fortran listing.

load
: With this form, the subsystem will begin by telling you the number of errors detected by the loader while loading a program.

user
: This form is for listings which do not conform to one of the above standard styles. If you answer "user" to "form?", SCAN will next ask the question "code?". A code is a sequence of characters which mark the beginning of lines you want to look at. For example, if you are looking at the output of a program which marks the beginning of important lines with "*****", you would answer

  ```
  code?*****
  ```

to this question. From this point on, SCAN would only look at lines beginning with the given code. Most of the time, you do not want a special code; thus you simply reply to the "code?" question with a carriage return.

dump
: This is the same as the "user" form, except that you will not be asked for a "code".

The next question SCAN asks is "edit?". SCAN wants to know whether you want multiple blanks squeezed into a single blank or not: answer "y" for blank suppression, and "n" to leave the output the way it is.

Now that SCAN knows what kind of a file you're looking at and how you want the output to be presented, you are ready to begin the actual scan of the output. SCAN will prompt with a "?", asking what you want it to do. There are several possible commands you can give.

print <n>
: This prints <n> lines of output, beginning at the current line. (The current line is line one when you first begin scanning a report code.) Unlike text editors like FRED, the "print" command does not change the current line. Thus saying "print 20" twice will print the same twenty lines of output twice. If no number <n> is specified, only the current line is printed.

find/<string>/
: This finds an occurrence of <string> and prints the line where <string> occurs. For example, "find/error/" will find a line where the word "error" appears. The search begins at the current line and goes to the end of the output; unlike text editors like FRED, the search does not wrap around from the bottom of the listing to the top.

space <n>
: This advances the current line by <n> lines. Thus

```
print 20
space 20
print 20
```

will print twenty lines, space down to the end of those twenty lines (remember, "print" does not change the current line), and then print the next twenty lines. Thus the given commands print 40 consecutive lines.

line <n>
: When SCAN prints lines of output, it always prints a line number alongside the output line. "line <n>" will set the current line to line number <n>. Thus "line 1" will set the current line to the first line of output. It is a common practice to use "line 1" before a "find" command so that you can search for a string throughout every line of output. If you do not use "line 1", you will only search from the current line to the end of the buffer.

errors
: When using the "gmap", "fort", or "load" forms, the "errors" command will list the errors found in the output listing.

done

This gets you out of SCAN and back to JOUT's "function?" question.

There are a number of other options that can be used with JOUT's SCAN function; there is even an entire SCAN subsystem that takes a similar syntax to JOUT's SCAN. For further information, see "expl scan".

8.5 JABT

The JABT TSS command aborts a running batch job. Just say

```
jabt snumb
```

at TSS system level and the batch job with the given snumb will be aborted. Note that the userid on the $USERID card of the given job must be the same as the userid of the person requesting that the job be aborted.

8.6 JSTS

The JSTS TSS command gives the status of a batch job. Saying

```
jsts snumb
```

at TSS system level will give you the status of the job with the given "snumb". The status message comes in the same format as the output from JMON.

8.7 BW

The BW TSS command lists jobs that are in the system. It is able to find jobs that are waiting in queues, jobs that are executing, and jobs that are waiting in SYSOUT to be printed. The command has the form

```
bw userid
```

where "userid" is the user whose jobs you wish to list. If you don't specify a userid, BW will list every job submitted from your own userid.

8.8 LSTWT

The LSTWT TSS command is much like the BKDR command, except that it lists the batch jobs that are waiting to be printed. It also lists the batch jobs that are waiting for JOUT or remote services.

9. Miscellaneous Notes

In this section, we will discuss a few other JCL cards that you might find useful. The final subsection of this guide presents a short glossary of terms that you might come across if you read any of the GCOS8 manuals.

any of the GCOS8 manuals.

9.1 The $TAPE Card

The $TAPE card is used to define files which are located on magnetic tape. The general format for the card is

```
$       tape    fcode,lud,multi,ser,reel,name,class,den
```

where "fcode" is the file code of the tape file and "lud" is the LUD for the file, possibly including a disposition code. The other arguments of the card are described below.

multi
: This indicates that two tape drives are to be allocated for this file code because the specified tape file extends over several reels of tape. This can save a lot of rewind time on long multi-reel jobs.

ser
: This is the five-character serial number of the desired tape. This number tells the operator which tape to mount for you. Normally the system will check to make sure that the operator has mounted the right tape; however, if you specify a "ser" of 99999 this checking does not take place.

reel
: If the tape you want has more than one reel, the "reel" argument gives the sequence number of the first reel you want mounted.

name
: When you are loading a Fortran program using the $EXECUTE card, this is the name of the file which you want to read or write on the given tape. In most other situations, this is the name that appears on the paper label on the outside of the tape you want. Specifying this name makes it easier for the operator to locate your tape. "name" can be a maximum of 12 characters long.

class
: Not applicable.

den
: This argument specifies the density of the tape. Possible values for "den" are

  DEN2
  : 200 Bits per inch.

  DEN5
  : 556 BPI.

  DEN8
  : 800 BPI.

  DEN9
  : instructs the system to use the default tape density for the site. It is generally safer to specify your own tape density rather than rely on the site default.

  DEN16
  : 1600 BPI

1600 BPI.

Below we give some examples of $TAPE cards.

```
$       tape    in,x1s,,12345,,yetis,,den16
$       tape    ot,f3d,,99999,,scratch,,den8
```

Normally tapes are mounted without write rings. A write ring is a little plastic ring on the back of the tape reel which must be present if anything is to be written on the tape. If you want the operator to mount your tape with a write ring, you must send the operator a message to this effect. This is done with a JCL card of the form

```
$       msg2    1,please mount tape 12345 with ring
```

This card should come immediately after the JCL card which defines the activity in which the tape is needed (e.g. $EXECUTE). Then when the batch processor is ready to execute that activity, the message will be sent to the operator. Note that it is a polite practice to include such a message to the operator for every tape you want mounted, whether it needs a write ring or not. Such messages make sure that the operator knows exactly what you want.

9.2 The $ETC Card

The $ETC card is used to continue argument fields from one JCL card to the next. You can break the argument fields at any natural breaking point (e.g. a comma or a slash) as in

```
$       prmfl   ot/x15,w,s,
$       etc     alibaba/cat1$pswd1/
$       etc     outfile$pswd2
```

Most of the time, of course, you should be able to get everything you want on one line.

9.3 Saving Money with $MSG3

Under many charging systems, you receive discounts on jobs run during low-use periods (e.g. in the wee hours of the morning). You can arrange to have jobs run at such times by using the $MSG3 card. The simplest format of this card is

```
$       msg3    yymmdd/hh:mm
```

where "yy" is the year, "mm" the month, "dd" the day, "hh" the hour, and "mm" the minute that the job should begin execution. For example,

```
$       msg3    811225/02:00
```

included at the beginning of your job will cause it to be held until two o'clock in the morning of December 25, 1981.

For further details about $MSG3, see "expl cc msg3".

9.4 A Small Glossary

9.4 A Small Glossary

If you are going to use batch to any great extent, you will eventually find yourself reading various GCOS8 Reference Manuals. In this small glossary, we will mention several terms that you will possibly encounter during the course of your reading. Our explanations here will be very brief and lacking in detail; still, we hope that what we say here will give you some idea of what certain terms mean. For the specifics, you will have to depend on the manuals themselves.

FCB
: An FCB is a File Control Block. This is the block of memory where a program stores all the information it needs in order to keep track of one of the files it's using. The FCB is mainly a Fortran construct; many other languages have different ways of keeping track of their files.

GCOS
: In case you were interested, it stands for General Comprehensive Operating System.

PAT Segment
: PAT stands for "Peripheral Attribute Table". This table is stored in a segment of memory, in order to keep track of the peripheral devices and files being used by a program.

privity
: Certain operations in batch may not be performed by unauthorized users. These operations correspond roughly to "privileged" operations in TSS. Any batch job wishing to execute one of this privileged operations in batch must include a $PRIVITY card in the activity which will perform the operation. The system will then ask the operator if it is all right to consider your job privileged. If the operator gives approval, the activity will be executed; otherwise, your job will be aborted. If you attempt to execute a privileged operation without receiving privity from the operator, your job will also be aborted.

report codes
: Whenever a line is written to a SYSOUT file, the line is accompanied by an octal number between zero and 63 (octal 077). This number is called a report code and is used by SYSOUT to separate output listings. After all, most lines of printer output are sent to a single file with file code "p*"; the system has to have a way to untangle the various listings and separate them in the proper way. Report codes help the system in this separation process.

  If your program is written in Fortran, the report code associated with a line of output is just the unit number of the output unit in octal. Thus if the line was written to unit 42, it would have report code 52 (since decimal 42 is octal 52). B programs can set the report code of an output unit using the SET.RC library function. Other languages either use default report codes or offer similar ways to set your own report code.

spooling
: When a program sends output off to be printed, the output does not go directly to the printer. Instead, it is passed to the SYSOUT program which organizes the output before printing. It does this through the process of "spooling". This means that the output is stored on disk until SYSOUT decides that it can be printed. SYSOUT waits for all the printer output to be generated by a batch job before it will send any of it to the printer; that way you get a single

listing instead of a bunch of little listings from every activity in the job. Thus all printer output must be spooled after each activity, until the final activity of the job has finished. Once this has happened, SYSOUT puts all the spooled output into the queue of jobs waiting to be printed. Batch printing jobs alternate with TSS printing jobs; thus SYSOUT will print a batch job, then a TSS job, then another batch job, and so on. There are usually fewer batch jobs awaiting print than TSS jobs (since there are fewer batch users) and therefore batch jobs usually get printed faster.

swapping

The GCOS system runs many jobs all at the same time. Usually there are so many jobs running in the system that there isn't enough memory to hold them all. Thus at any given time some jobs will be stored in memory and working, while other jobs are stored off on disk waiting for their turn to use the memory. The working jobs are "swapped in" while the waiting jobs are "swapped out". Being swapped out is not the same as waiting for execution; a job that has been swapped out has already been executing for a while. While swapped out, it is frozen in the middle of execution; when the core allocator finds memory space for the program, it will be swapped in and will start executing again until the next time it is swapped out.

Appendix A: Useful Explain Files

Below we list some explain files that contain useful information for batch users.

expl abort

Explanations of abort codes used when batch jobs abort.

expl b batch

How to use B programs in batch.

expl batch queues

A discussion of system scheduler queues.

expl batch status

A brief description of each job status message issued by JMON.

expl bw

The BW command lists all the batch jobs in the system that were submitted by a given userid.

expl cc

Brief explanations of each of the GCOS8 JCL control cards.

expl conv

Information about the CONVert subsystem (CONV is closely related to JRN).

expl fortran batch

How to compile and execute Fortran programs in batch.

expl jabt

The JABT command aborts a batch job.

expl jmon

The JMON subsystem monitors the progress of a batch job.

expl jout

The JOUT subsystem can be used to examine the SYSOUT output of a batch job.

expl jrn

The JRN command will run a job in batch.

expl jsts

The JSTS command will return the status of a batch job.

expl lsta

The LSTA command can give information about everything that's happening in the batch world

expl lstw

The LSTW command lists what's happening with SYSOUT.

expl pascal batch

How to compile and execute Pascal programs in batch.

expl pascal batch options

This lists the possible options that can be specified for the batch version of the Pascal compiler.

expl queue

The QUEUE command lists the jobs that are waiting in the system scheduler queues.

expl scan

How to use SCAN and the SCAN function of JOUT.

Appendix B: System File Codes

Below are listed the file codes most commonly used by system programs. For a complete list, see the GCOS8 JCL Reference Manual.

**

location of program for activity

b*

object deck produced by compiler

g*

source for GMAP assembler

h*

core image produced by loader

i*

default file code for non-JCL found in input stream

*l

site-supplied system library

l*

GCOS8-supplied system library

p*

SYSOUT file

q*

system-loadable file

r*

file containing loader directives

file containing loader directives

s*
: source for compiler

In addition to the above system file codes, there are several other file codes which have special meanings in some contexts.

bb
: a standard library for Pascal

bl
: another Pascal library

cz
: contains command lines for Pascal programs, B programs, TF, etc.

pl
: another Pascal library

pm
: contains the abort file from a B program