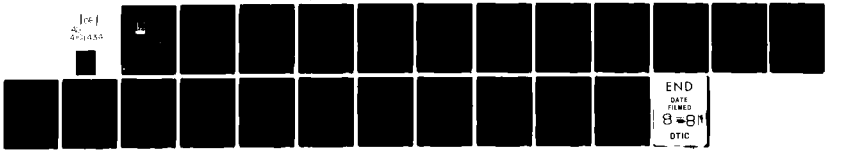
AD-A101 434 ARMY ENGINEER WATERWAYS EXPERIMENT STATION VICKSBURG MS F/G 9/2
TWO-DIMENSIONAL GRAPHICS COMPATIBILITY SYSTEM, VERSION 3.0. HON--ETC(U)
JUL 78

UNCLASSIFIED DOD/DF-81/005B NL



END
DATE FILMED
8-81
DTIC

WATERWAYS EXPERIMENT STATION

Vicksburg, Miss.

A

HONEYWELL COMPUTER OVERLAY SCHEME FOR 2-D GCS

for the WES Automatic Data Processing Center

(6)

TWO-DIMENSIONAL

GRAPHICS COMPATIBILITY SYSTEM

Version 3.0

HONEYWELL COMPUTER OVERLAY SCHEME

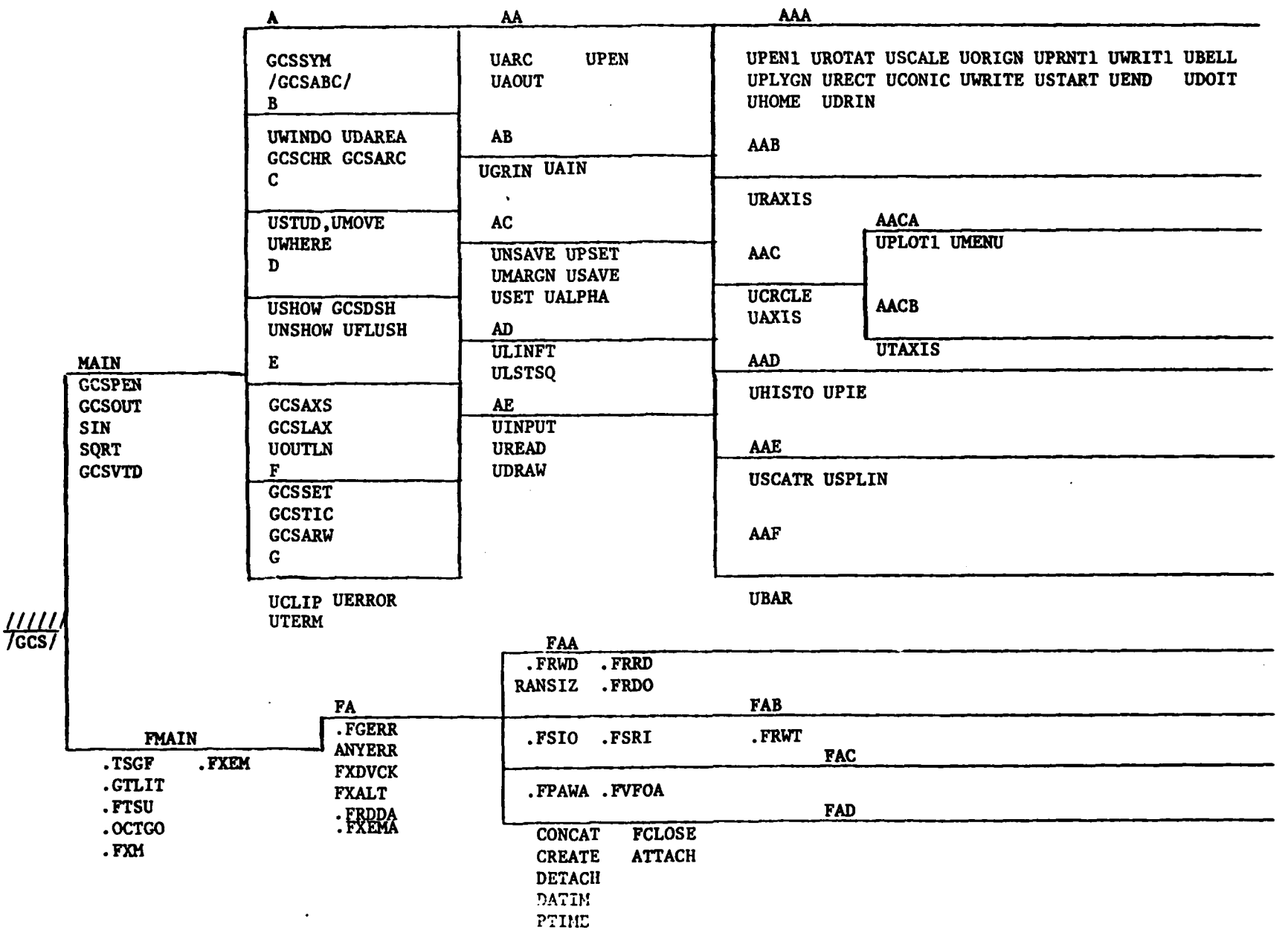
(18) DoD/DF (19) [illegible]

(12) 23

(11) 5 Jul[redacted]78

CONTENTS

GCS TEKTRONIX OVERLAY SCHEME



DESCRIPTION OF THE GCS OVERLAY SYSTEM

During the spring of 1973, the Graphics Compatibility System developed by ISD was nearing the final stages of debugging and showed great promise. One persistent problem, however, was poor response time during a GCS execution due to the cumbersome size of the GCS library. To meet this problem, a pre-overlaid version of the GCS library and a run-time executive to coordinate swapping has been developed. Primary goals of this effort were:

(1) the entire GCS library must be available,

(2) the core requirements should be modest,

(3) the user need not specify any explicit overlaying, and

(4) the user must be able to write, debug, and run his routine under time-sharing.

The balance of this paper describes this technique. Section I discusses the concepts involved. Section II describes the GCS implementation in detail.

CONCEPTS

The use of any structured overlay technique requires the programmer to impart to his routines some kind of structure. In the case of an unoverlaid load that structure is linear. In the case of a conventionally overlaid program, a tree structure is used with the root of the tree containing the focus of control. In such a program, the main routine, sometimes called the "driver", resides in the root segment and calls lower-level routines which reside on subordinate segments.

It is also possible to impart a tree structure to a subroutine library using a converse philosophy. Such a library can be structured by putting the most basic, low-level routines in the root segment and putting higher-level subroutines on subordinate segments. We impose the rule that a subroutine may only call subroutines that either are on the same segment as the caller or on a segment on the path from the caller's segment to the root. If this rule is followed when the tree structure is designed, then any given routine in the library can function without worrying about which segments are in or out of core, provided only that, prior to starting the routine, the path from this given routine to the root is brought into core. Consequently, memory management within the subroutine library can be performed by passing briefly through a run-time Monitor prior to each library call.

The implementation of this scheme involves three distinst steps. The first is the design of the subroutine tree structure. If the structure is simple, the library can be loaded in a single GLOAD activity. The result of this first step is a pre-loaded, overlaid version of the library in H* format; this file is subsequently referred to as the H# file.

The second step is the writing of the Monitor. Two versions of this Monitor -- one for batch and the other for time-sharing -- have already been written and tested. This Monitor will perform without modification for any library produced according to the conventions outlined in this paper.

The third step is the writing of a Table which describes the particular structure of this library to the Monitor. All the symdefs of the library reside in this Table.

When the user's routine is loaded, the Table is brought in by GLOAD from a random user library to satisfy any symrefs for library routines. Similarly the Monitor, which also performs the functions of .SETU., is brought in by GLOAD. Only at run-time, upon calls to library routines, are segments

of the pre-loaded H# brought into core by the Monitor.

Experience with this technique has shown it to be feasible in providing effective memory management while localizing all memory management functions to these three steps. In particular, neither the users of the library nor the programmers who wrote the library routines had to modify their code.

The GCS Implementation

This section explains the implementation of a preloaded library in some depth using, as an example, a large graphics library for time-sharing.

The Graphics Compatibility System (or GCS) is a large collection of graphics subroutines, all written in ANSI Fortran . About fifty of these routines have names beginning with a "U", e.g., USTART, which are called by the user. Others are called only by other library routines; these number about 28 and have names beginning "GCS", e.g., GCSARC. All these routines communicate with each other via a labeled common region called "GCS". None of these routines use the Fortran I/O library, but the Fortran mathematical library is heavily used.

With this background in mind, consider the loading of the H# file and, in particular, the tree structure formed by segments //////,MAIN,A,B,C,D,E,F, G,FMAIN, and FA (refer to figure 1). Following our convention of calling only those routines on the path from the caller to the root, we have designed this structure so that a routine in segment C, for example, can only call routines in segments C,MAIN, and //////. In the case of this tree structure, the only element of segment //////, the core-resident segment, is the labeled common region, "GCS". This labeled common region must be core resident, since it contains global variables and may never be reinitialized. The two sons of ////// are MAIN and FMAIN. Segment MAIN contains the most basic GCS routines, along with the Fortran mathematical functions. While further segmentation of MAIN would be logically possible, the savings in core would be offset by the need for more frequent swapping. Segment FMAIN contains the most basic routines of the Fortran I/O library. Our tree structure will allow the GCS library and the Fortran I/O library to share the same core, while making both conventional I/O and graphics simultaneously available to the user. The sons of MAIN contain less basic routines than those in MAIN. Some call routines in MAIN, some are basic routines, but are infrequently called. Similarly the routines of FA and FB are less basic than those in FMAIN and contain those routines entered directly from the user's Fortran code. At this point, refer to the listings marked "LOAD TEXT" on the banner; the first GLOAD activity of this job. Note that it is a conventional GLOAD deck setup, but that a few things are unusual. First,

the NOSETU option is selected, for .SETU. will be pulled in when the user is loaded. Second, the specification of .CMN. as an entry point; this merely prevents GLOAD from refusing to proceeed for lack of an entry point. Third, libraries G2 and G1 are random user libraries on which the graphics routines are found. Room for two labeled common regions, .FLTX1 for FMAIN and GCS for MAIN, is reserved via the USE card. In segment MAIN a single object deck called TEKINT is included; its function will be discussed later. The routines to be put into each segment are determined by USE cards; they are found on libraries G2, G1, and L*. Since we are satisfied of the correctness of the GCS routines and want to conserve core, we prevent the loading of .FXEM. and .FIXT., two error message routines; GCSBLD, which is yet unimplemented, is also excluded. The last unusual feature is the inclusion of a labeled common region, GCSABC, in segment A; since GCSABC is merely a table of constants initialized via a BLOCK DATA subprogram; it may be placed on a non-resident segment. With these explanations, the reader should be able to follow the reports for the first activity without difficulty.

Now that we understand more of the nature of an H# file, let us examine the Table structure used to describe its structure to the Monitor. The Table describes a tree structure of physical segments, each containing several subroutines; it is assembled by expanding a set of macros. These macros and their expansion for the tree structure just described are shown in the listing marked *TEK INT* on the banner. The physical segments themselves are described via four-word LNODEs, which contain the BCD name under which the segment was loaded, a pointer used to describe the father of the segment, the number of subroutines in the segment, and several booleans. This information is sufficient to tell the Monitor how to pull in the segment from H#, how many subroutines to mark as "in core", and how this segment fits in with the tree structure. Immediately following each LNODE are the proper number of SNODEs, each of which describes one subroutine contained within the segment. The SNODE is a TRA instruction pointing to the actual core location of its corresponding routine. When the routine is on core, this TRA instruction simply routes him to the routine; when the routine is not in core, the address modification field of the TRA is set to fault tag modification, thus transferring control to the Monitor. The reader should convince himself that the Table shown in the Appendix does describes the tree structure in complete detail and also provides enough information to

restore any segment from the H# file.

Let us now proceed with a description of the Monitor. We specified that with the existence of a H# file and an appropriate Table, the Monitor could provide all necessary run-time memory management. The Monitor contains the symdef .SETU. and is accordingly brought into core by GLOAD. Upon receiving control in its .SETU. role, the Monitor performs ordinary setup functions, then sets the fault vector so that any fault tag faults will be handled by the Monitor in its memory management role. Upon any TSX1's to SNODEs that have the fault tag fault bit on, the Monitor will be given control and notified of the location of the fault. The Monitor then executes a three-step procedure to properly reallocate memory. First it visits every LNODE along the path from the one that causes the fault to the root segment until it reaches the first one presently in core, setting each SNODE in-core. Second, it visits every LNODE that represents a segment that must be overlaid and marks their SNODEs out-of-core. Third it physically restores each segment visited in the first step. The first time a given segment is restored, its location on the H# and the DCW word used to read it into core are both saved within the LNODE; subsequent restores are thus handled in a single physical read. After these three steps are done, the TSX1 may be completed successfully. The GMAP source of the TSS version of the Monitor is shown in the Appendix under the listing marked EXECUTIVE.

The basic workings and interactions of the H# file, the Table, and the Monitor have now been fully explored. The GCS implementation is slightly more sophisticated, however. Recall that, while the segmented library is not physically core resident, it does <u>appear</u> to be core resident so long as all library calls are directed through the Table. This fact was used to great advantage in the GCS implementation. Consider segments AA, AB, and AC in figure 1. If segment AA is in core and one of its routines, say UPEN, calls a routine in segment C, say GCSPSA, and does so through the Table, then the call will be effective. Similarly, due to our conventions, all routines in segments MAIN,A,B,C,D,E,F, & G are available to UPEN via the Table. In this fashion a second tree structure including segments AA, AB, AC, AD and AE constructed; each segment was loaded rather like a user who was aware of only the partial GCS library made up of segments MAIN and A,B,C,D,E,F, and G. To accomplish this, the Table for segments MAIN and A,B,C,D,E, F and G were

included in segment MAIN. In activities 2, 3, 4, and 5 of the job marked LOAD-TEXT, each of these four segments was loaded and added to the H# file. Each of these activities is very similar to the first activity with the following exceptions. First the LOWLOAD card has been changed to make room for segments A,B,C,D,E,F & G. Second a number of EQUATE cards were used to direct subroutine calls through the Table in segment MAIN. Third the OPTION SAVOLD was used to force these segments onto the H# file. The reader should convince himself that, if segment MAIN and segment AA are both in core, then calls from routines in AA to any routines im segments MAIN or A,B,C,D,E,F or G will function correctly. For purposes of calls from the user to routines in segments AA, AB, AC, AD, or AE, a pseudo-segment, R, was included in the Table to complete a second independent tree structure made of segments R,AA,AB,AC,and AD. This Table is the object module TEKINT that we mentioned in our discussion of the first activity.

A third tree structure, including segments AAA, AAB, and AAC, was then loaded in activities 6, 7, and 8 of the job marked LOAD TEXT. Once again it was important that routines in segment AAA be allowed to call routines in segments AA, AB, and AC. As mentioned above, as long as these calls were directed through the Table in segment MAIN, all the routines in segments MAIN, A-G, and AA-AE would indeed appear to be core resident. This slight extension of the basic concept to allow for multiple tree structures was instrumental in achieving a great compaction of core requirements. In GCS, we were able to load about 22K worth of Fortran code into 8K of core and managed to provide the Fortran I/O library alongside it with no additional core requirement. Our experience shows that the volume of disc I/O required to provide the necessary swapping is quite modest, although an ill-structured H# file might necessitate disastrously heavy swapping.

To complete this description of the GCS implementation, consider the Table that must be brought into the user's area. As shown on the listing marked LOAD-XTEK, the user's Table describes a single tree structure. Segments AAA, AAB, AAC, FMAIN, and FA are arranged in the obvious way. Segment MAIN, however, lists on its SNODEs all the routines in segments MAIN--A-G, and AA-AE. These SNODEs point to the actual entry points for the routines physically on segment MAIN, but they point to the SNODEs of the Table in MAIN for those routines in A-G and AA-AE. this insures that our assumption about the presence of MAIN in the discussions about segments AA-AE and AAA-AAF would be valid.

TEXEC, THE TSS OVERLAY MONITOR

Run-time memory management is handled by a Monitor which is activated by the fault tag fault whenever a desired segment is not in core. This Monitor is written in GMAP and is invarient with differing overlay structures. The code occupies the first module of a Source/Object Library pair called GRAFD/SUPPORT/KSTAR (RSTAR). The action of this Monitor will be examined in its two roles: that of a special .SETU. and that of a fault tag fault handler.

TEXEC At Initialization

There are two reasons for including .SETU. functions in TEXEC. The first is essential: the fault vector must be modified to work correctly in an overlaid environment. The second is for the sake of efficiency. The code used in .SETU. may be reused as buffer space by TEKEC upon subsequent activation.

TEXEC sets three fault vector cells in a non-standard fashion. The divide-check and overflow cells may not call .FXEM., since FXEM is overlaid and may not be in core at the time of the fault. The third cell is the fault tag fault cell, which is set to trap to the Monitor.

One of the functions of .SETU. is the construction of an LGU and FCB's. This code was cleaned up and laid out in a fashion to facilitate later use as buffer space. The sixty-four words beginning at .SETU. later comprise a disc input buffer for use by the Monitor. The eight words at .SETU. + 64 later are used for storage of the user's registers during execution of the Monitor. The forty-six words at TTYBSF are used for FCB's.

To reiterate, the really crucial thing here is the setting of the fault tag fault cell to point to .VM.EX, the run-time Monitor.

TEXEC Upon the Fault Tag Fault

Upon activation, the Monitor first saves the user's registar and resets the fault tag fault cell. The IC at the time of the fault is retrieved from the fault tag fault cell; IC-1 is now stored as the return address from the Monitor.

This IC is now used to determine the needed segment (see Figure 2). Now the address IC-1 points to the faulting SNODE. The LNODE is located by back-tracking up the list of SNODEs until a word with bit 18 clear is found. All SNODEs are TRA instructions and have bit 18 set; all LNODEs end with an IOTD DCW and have bit 18 clear. Having found the LNODE, this LNODE is marked as "FIRST" (by storing its address in location FIRST) and is also made the "current" LNODE (x2 is used to point to the current LNODE). We will now mark our current LNODE and all his ancestors as being in core. There are three differences between an LNODE marked in core and one marked swapped. (1) The zeroth bit of LNODE+2 is set if the LNODE is in core. (2) All address modification fields on SNODEs subsequent to the LNODE are cleared. (3) The LINK field of the LNODE is set to that (unique) descendent that is also in core (this descendent's address is kept in X1). If there is no in core descendent, then a null pointer, λ, is set in the LINK field. This is the case for the first LNODE. So starting with the "first" LNODE and proceeding to the current LNODE's ancestor until that ancestor is already marked as in core; this LNODE is termed the "LAST" LNODE (X4 is used to point to "LAST" LNODE). At this point, all the segments (X3 is used to point to the current LNODE's ancestor) that we have put marked as in core are in a linked list proceeding from the current LNODE which is already physically in core. Later on, we will run back through this list physically hauling in the segments.

During the second phase of Monitor response, segments that are in core, but must be logically (and perhaps physically) swapped out, are located, marked out of core, and, if writeable, swapped out. Beginning with the in core descendent of the "LAST" LNODE, and continuing with his (unique) in core descendent while there is an in core descendent, this descendent is marked out of core. (If the segment is "writeable", then it is written onto "tempfile" and marked as "written"; this feature has not been tested).

The third phase of Monitor response physically reads in those segments marked in core during the first phase. Beginning at the "LAST" LNODE and

proceeding to its in core descendent until reaching the "FIRST" LNODE, the segments are read into core. If the segment is "written" then it must come from "tempfile"; if it is a library segment (it is this case in GCS) then it must come from the H# file; otherwise it comes from the users H* file. If the relative block number field of the LNODE is non-zero, then the relative block number and the IOTD stored in the LNODE are used as the seek address and DCW in a disc read. If this field is zero, however, a three-part process is performed: First the Catalogue Block(s) of the file are searched for a segment entry whose name is the first word of the LNODE. If found, this entry contains the relative block number of the Control Block of that segment. This block is now read. It contains the IOTD for the actual segment read and the relative block number of the segment is one greater than that of the Control Block. This relative block number and IOTD are stored on the LNODE for future use.

Upon reaching the "FIRST" LNODE, the user's registers are restored and the faulting instruction in re-initiated. This time, of course, the fault tag modification has been cleared and the Monitor remains dormant until reactivated by another fault tag fault.

Figure 2

```
LNODE     BCI     1,LKNAME                  43 42 45, 21 44 25
          ZERO    FATHER, 4 *2              (father), 00 20 00
          VFD     1/0, 1/0, 2/1, 32/0       04 00 00  00 00 00
          ZERO                              00 00 00  00 00 00
SNOD1     TRA     SUB1, F                   (sub1)   , 710 040
SNOD2     TRA     SUB2, F                   (sub2)   , 710 040
SNOD3     TRA     SUB3, F                   (sub3)   , 710 040
SNOD4     TRA     SUB4, F                   (sub4)   , 710 040

LNODE     MACRO
          BCI     1, #1
          VFD     18/#2, 8/#3,  10/0
          VFD     1/#4, 1/#5, 2/#6, 032/#7
          IOTD    #8, #9
          ENDM    LNODE
```

#1 SEGMENT NAME IN BCD; used to search the H# catalogue
#2 LINK; either direct ancestor or the (unique) descendent in core
#3 NOSUBS; the number of SNODES that follow
#4 CORE; set iff this segment is in core
#5 WRITABLE; set iff this segment is writable and must be swapped out upon overlay.
#6 LIBRARY; set iff segment is on the library H# file; clear iff segment is on the user's H* file.
#7 the relative block number (seek address) of the segment on the H# or H* file.
#8,#on IOTD DCW word

```
SNODE     MACRO
          SYMDEF   #1
          VFD      018/#2, 018/710040 (TRA #2, F)
          ENDM
```

#1 NAME; the subroutine name
#2 ADDRESS; in octal

Maintenance of the Overlaid GCS Library

In order to provide an efficient run-time environment for GCS routines, an overlay scheme has been developed that overlays the GCS library -- not the user's program -- and controls calls to GCS via (1) a run-time Monitor, (2) a descriptive table, and (3) a core-image subroutine library. This paper discusses the files needed to maintain these three elements and the process required to maintain these files. The Tektronix 4010 version of GCS will be used as a platform for discussion throughout.

When the user's routine is loaded, the run-time Monitor and the Table mentioned above are brought into core from a random user library called TEK. The Monitor is fixed, but the Table must reflect the structure and exact octal addresses of important routines stored on the core-image subroutine library TEKHS. TEK is built by means of a job on file BDTEK operating on files KSTAR and RSTAR. TEKHS is built by means of a job on file BDTEKHS operating on files TEKINT, TEKEX, TEKEQ, TEKAA, TEKAAA, GCSLIB, FORTLIB, and TEKLIB. Of these files, KSTAR, RSTAR, and FORTLIB should remain fixed despite any change to either the GCS source or the desired structure of TEKHS. Changes in GCS source files are relected in random user library files GCSLIB and TEKLIB via jobs BDGCSLIB and BDTEKLIB operating on the GCS ascii source files. Changes in the sizes of any of these routines or in the desired structure of TEKHS will necessitate changes in some of TEKINT, TEKEX, TEKEQ, TEKAA, TEKAAA. TEKINT is built by means of a job on file BDTEKINT. The others are built via text editor.

Lest a harried soldier or civil servant suspect this multiplicity of files puts the situation out of control, the process of maintenance will be examined in more detail. Suppose the ascii source files have been changed. The jobs on BDGCSLIB and BDTEKLIB are run; they (1) compile the new source onto a merged C* file, (2) merge them with a $ENDEDIT card and other C* files via UTILITY, (3) create an R* through an OBJECT-INITIALISE FILEDIT run, and (4) create a random user library via RANLIB.

Now suppose these random user libraries are built; our new source code is reflected in the unoverlaid version of GCS and we are ready to rebuild our core-image library. At this point the structure of the library must be decided; that is, the decision of which routines go into which links must be made. TEKINT must be rebuilt if any structural changes are to occur in the

A or AA levels. The new TEKINT must reflect the new structure of these levels precisely; the actual octal addresses in the SNODEs of TEKINT are irrelevant at this stage.

With this step performed, the routine BDTEKHS can be run to obtain a load map. The structure of the library can be modified by changing BDTEKHS itself. TEKEQ, TEKAA, and TEKAAA must be edited so that (1) the symdefs that appear in EQUATE cards follow the structure of the library (2) the value on the LOWLOAD card avoids an overlapping of Level A with AA or AA with AAA, and (most tedious: 3) the value on the EQUATE cards must precisely reflect the address within link MAIN at which either the symdef itself (as in GCSPEN) or the SNODE (as in UPEN) corresponding to the EQUATEd symbol has been loaded. Several iterations of this step are usually required before all the octal address in the block common list following each routine in levels AA and AAA reflect the correct address in link, MAIN. It must be remembered that these addresses on the load map are actual octal addresses and that the numbers on the EQUATE cards are decimal offsets from location 64. No amount of documentation can make this step anything but a tedious clerical chore.

With this step behind us, life becomes easier. Now the octal addresses in TEKINT must be made correct and job BDTEKINT rerun. The job BDTEKHS is now rerun to incorporate these accurate addresses. This job will result in a proper load, for the size and structure of TEKINT have not changed!

The core image library is now fixed. Job BDTEK is now used to update the Table of pointers into this core image library. Both its structure and the actual addresses must be made to reflect the structure and addresses found on the load map of the core image library.

Finally, the block common sub-program TEKCOMN must be examined to insure that its size is large enough to reserve the space between location 64 and the user's program needed for any combination of segments from the core image library.

GLOSSARY OF FILE NAMES

TEK	Random User Library Containing Monitor & Table
TEKHS	Core-image library of GCS subroutines
KSTAR RSTAR	Source & Object Library pair containing the Monitor and Macros for the Table; used to build TEK
TEKINT	A Table describing GCS routines in level A and AA to levels AA and AAA
TEKEX	A select file of control cards; basically a $ EXECUTE, $ LIMITS, etc
TEKEQ	A select file of $EQUATE cards specifying the locations of routines and SNODEs in link MAIN to levels AA and AAA
TEKAA	A select file containing a $ LOWLOAD card and a few $ EQUATE cards perculiar to level AA
TEKAAA	A select file similar to TEKAA, but for level AAA.
GCSLIB	Random User Library of Device-independent GCS routines.
TEKLIB	Random User Library of Tektronix versions of device-dependent GCS routines.
FORTLIB	Random User Library of Fortran I/O subroutines modified to behave correctly in the overlaid environment.
TEKCOMN	A Fortran source block data sub-program concatenated to the user's program to reserve space needed by GCS at run-time.
BD---	A series of cardin files that, when submitted as a job will build the file whose name is the suffix of the name of the cardin file, e.g., BDTEKINT builds file TEKINT.

Use and Maintenance of the GCS Subsystem

The accessing of files and typing of a suitable run command successfully to run a GCS program requires a tedious and error-prone task for the GCS user. Changes in the names of files would further require thorough communication among all GCS users. To meet these needs, a special subsystem is provided under CMDLIB to access files and pass a suitable run command to the RUNY subsystem. A HOLD command is also simulated to prevent interference of spurious messages with graphical output. Finally, all GCS files are deacesses upon the completion of the GCS run. We discuss in this paper the use of the subsystem and discuss its implementation in order to facilitate maintenance.

Use

At the SYSTEM or * level, the user may type GCS to simply run his current source GCS routine, just as he would type RUN for ordinary Fortran. As described in the Fortran manual, however, the user may also type more involved run commands in order to select non-standard compiling or loading options or to select source decks other than his current file. For instance, if the user wished to run his current file concatenated with a c-star file under alt-name FIL2 and select the optimization option in the compilation of his current file, he would type:

```
GCS  *;FIL2=(OPTZ)
```

If he wished to run a third file called FIL3 alone and invoke a user library called MYLIB and select the formatted option of source file interpretation on his compilation, he would type:

```
GCS FIL3=(FORM)MYLIB
```

This last example points out one of three differences between the GCS command and the Fortran run command, which is that the ULIB option during loading is standard, since GCS user libraries are always needed for a GCS load. The second difference is that h-star format files may not be run via the GCS command. It is suggested that users save their important routines at the c-star level for quick rerun. The last difference is that the BCD option is not useful, since all the GCS library is pre-compiled in the ascii mode.

Having issued his GCS command, the user will be asked which version of GCS he wishes to run. If the version name begins with an "X" or there is no overlaid version for the graphics device indicated, then he will receive an unoverlaid version of GCS. Otherwise the standard overlaid version is provided.

Implementation

The GCS subsystem is a moderately complex GMAP routine. Its action covers four phases:

a. Parse Command

b. Access Files

c. Simulate Fortran run command

d. Deaccess Files and return

Parse Command

GCS begins by simulating a "HOLD" command, specifying the "FORT" compiler and establishing a break vector, all via the DRL SETSWH. The next action is to retrieve the user's GCS command via the DRL KIN; the command is put into a buffer and a tally BTAL is set pointing to it.

Next the user is asked the version and device he desires. If, for example, in response to the query DEVICE?:, he responds TEK he has chosen the (overlaid) TEKTRONIX version of GCS. For any devices for which an overlaid version exists, this overlaid version is the default. If however, he had responded XTEK, or if no overlaid version existed for the device TEK, then the unoverlaid version will be supplied. If the version chosen is overlaid, a simple "GCS" command will be transformed into the command;

```
RUNH  STUBABC;*= (ULIB) LIBR
```

If the version chosen is unoverlaid, the simple "GCS" command will be transformed into the command:

```
RUNH STUBABC;*= (ULIB) LIBR2;LIBR
```

In both cases, explicit options chosen by the user will be merged into the run command. For example if, as in part A, the user had typed GCS *;FIL2 = (OPTZ), this command would be transformed into RUNH STUBABC;*;FIL2 = (ULIB,OPTZ)LIBR. This merging of standard and user-selected options is performed by some fairly involved GMAP code. Once this transformation is complete, a tally LTAL is set poínting to the new run command and the next phase is entered at symbol GETS.

Access Files

The user's choice of version determines the action take by phase two. If the version selected is overlaid, the appropriate core-image library is selected from table HLIST and accessed via subroutine ACCESS. Next the random user library, if overlaid, or the device-dependent random user library, if

unoverlaid, is selected from table EXLIST and accessed via subroutine ACCESS. If the version selected is unoverlaid, then the device-independent library is now accessed via ACCESS. The last permanent file accessed is STUBABC, which is the GCSABC labeled common region if unoverlaid or the TEKCOMN labeled common region if overlaid. Subroutine relies on the DRL FILART and gets all its files from one catalogue.

Finally, it is determined whether or not the user has currently defined a *SRC file; if he has not, the OLDN subsystem is called to establish one. The third phase is now entered at symbol ZAP.

Simulate Fortran Run Command

Using DRL PSEUDO and the tally LTAL, the transformed run command is now set into the UST. Next the RUNY subsystem is called via the DRL CALLSS; if, upon return, bit 13 is on, this phase is terminated. Otherwise the .YPO subsystem is called via DRL CALLSS to load the user and run him. This RUNY-test bit 13-.YPO sequence is copied from the response of TSS to the run command outlined in the command language of module TSSA. Finally, the fourth phase is entered at symbol FINIS.

Deaccess Files

The fourth phase uses the DRL RETFIL to remove all GCS files from the AFT. It then uses the DRL RSTSWH to simulate a "SEND" command.