MASTER COPY
DO NOT REMOVE

CGTM No. 171
December 1976

Revised:
January 1978

THE SLAC UNIFIED GRAPHICS SYSTEM

INTERNAL OPERATION AND MAINTENANCE MANUAL

ROBERT C. BEACH
COMPUTATION RESEARCH GROUP
STANFORD LINEAR ACCELERATOR CENTER
STANFORD, CALIFORNIA 94305

TABLE OF CONTENTS

SECTION	DESCRIPTION	PAGE
1	INTRODUCTION	1
1.1	A BRIEF DESCRIPTION OF THE LIBRARIES	1
1.2	A BRIEF DESCRIPTION OF THE CONTROL BLOCKS	2
1.3	A DESCRIPTION OF A PROGRAM IN EXECUTION	3
2	A DETAILED DESCRIPTION OF THE CONTROL BLOCKS	5
2.1	THE MAIN COMMUNICATION AREA (MCA)	5
2.2	THE DEVICE-DEPENDENT AREA (DDA)	8
2.3	THE ATTENTION QUEUE ELEMENT (AQE)	10
3	A DETAILED DESCRIPTION OF THE DATA SETS	12
3.1	SOURCE DATA SETS	12
3.1.1	THE LIBRARY WYL.CG.RCB.UGSOURCE	12
3.1.2	THE LIBRARY WYL.CG.RCB.UGCOMMAC	13
3.1.3	THE LIBRARY WYL.CG.RCB.UGPTMMAC	13
3.1.4	THE LIBRARY WYL.CG.RCB.UGPLOMAC	13
3.1.5	THE LIBRARY WYL.CG.RCB.UGPLPMAC	13
3.1.6	THE LIBRARY WYL.CG.RCB.UGRUNMAC	14
3.1.7	THE DATA SET WYL.CG.RCB.UGCOMCNL	14
3.1.8	THE DATA SET WYL.CG.RCB.UGPTMCNL	14
3.1.9	THE DATA SET WYL.CG.RCB.UGPLOCNL	14
3.1.10	THE DATA SET WYL.CG.RCB.UGPLPCNL	14
3.1.11	THE DATA SET WYL.CG.RCB.UGRUNCNL	15
3.1.12	THE LIBRARY WYL.CG.RCB.UGPL1DCL	15
3.2	OBJECT DATA SETS	15
3.2.1	THE LIBRARY WYL.CG.RCB.UGPTMLIB	15
3.2.2	THE LIBRARY WYL.CG.RCB.UGPLOLIB	15
3.2.3	THE LIBRARY WYL.CG.RCB.UGPLPLIB	15
3.2.4	THE LIBRARY WYL.CG.RCB.UGRUNLIB	16
3.3	OTHER DATA SETS	16
3.3.1	THE DATA SET WYL.CG.RCB.UGPGMDOC	16
3.3.2	THE DATA SET WYL.CG.RCB.UGINTDOC	17
3.3.3	THE LIBRARY WYL.CG.RCB.UGTESTPG	17
3.3.4	THE LIBRARY WYL.CG.RCB.UGRUNMOD	18
3.3.5	THE LIBRARY WYL.CG.RCB.UGEXFILE	18
3.4	FORMAT OF THE BACKUP/DISTRIBUTION TAPE	18
4	THE INTERNAL FORMAT OF A GRAPHIC ELEMENT	20
4.1	POINT DATA	21
4.2	LINE DATA	21
4.3	TEXT DATA	21
4.4	DEVICE-DEPENDENT DATA	22
5	THE STRUCTURE OF THE MODULES	23
5.1	MODULES INDEPENDENT OF THE DEVICE-DEPENDENT CODE	23
5.2	MODULES WHICH ENTER THE DEVICE-DEPENDENT CODE	25
5.3	INTERNAL MODULES	26
5.4	DYNAMICALLY LOADED MODULES	27
5.5	UTILITY MODULES	28
5.6	ERROR MESSAGE MODULES	29
5.7	RECURSIVE ERROR PROCESSORS IN FORTRAN	31

TABLE OF CONTENTS

SECTION	DESCRIPTION	PAGE
6	EXTENDING THE UNIFIED GRAPHICS SYSTEM	33
6.1	ADDING A NEW GRAPHIC DEVICE TO THE SYSTEM	33
6.2	ADDING A NEW PROGRAMMING LANGUAGE TO THE SYSTEM	34
6.3	TRANSFERRING THE SYSTEM TO ANOTHER HOST COMPUTER	35
7	SPECIAL ALGORITHMS	37
7.1	SUBROUTINE UGXHCH	37
7.2	SUBROUTINE UG3DMS	39
7.3	SUBROUTINE UGCNTR	41
7.4	SUBROUTINE UGDXDY	45
7.5	SUBROUTINES UGPROJ, UGORTH, AND UG3TO2	45
7.6	THE LINE SCISSORING MODULE	48
7.7	THE LINE STRUCTURE GENERATOR	50
7.8	THE CHARACTER STROKE GENERATORS	51
8	NON-INTERACTIVE GRAPHIC DEVICES	53
8.1	THE CALCOMP DRUM PLOTTER (MODEL 1136)	53
8.2	THE CALCOMP MICROFILM PLOTTER (MODEL 1675)	54
8.3	DISPLAY FILES FOR THE TEKTRONIX 4013	54
8.4	DISPLAY FILES FOR THE GEN-COM 300-Q	55
8.5	THE VERSATEC ELECTROSTATIC PLOTTER (MODEL 1200A)	56
8.6	A PARTITIONED DATA SET PSEUDO-DEVICE	58
8.7	A SEQUENTIAL DATA SET PSEUDO-DEVICE	58
9	INTERACTIVE GRAPHIC DEVICES	60
9.1	THE IBM 2250 DISPLAY CONSOLE	60
9.2	THE TEKTRONIX 4013 DISPLAY TERMINAL	61
9.3	THE SLAC EXPERIMENTAL DISPLAY CONSOLE	62
10	NON-STANDARD GRAPHIC DEVICES	65
10.1	THE VERSATEC PLOTTER AT THE 40" BUBBLE CHAMBER	65
10.2	THE VERSATEC PLOTTER AT LASS	66
10.3	THE TEKTRONIX 4013 DISPLAY TERMINAL	66
10.4	THE SLAC SLAVE SCOPES	66
11	COMPARISONS WITH OTHER SYSTEMS	67
11.1	THE GRAPHICS COMPATIBILITY INTERFACE	67
11.2	DISSPLA	67
11.3	GINO-F	68
11.4	THE GRAPHICS COMPATIBILITY SYSTEM	69
11.5	THE GRAPHIC DISPLAY SYSTEM	69
11.6	THE INTEGRATED GRAPHICS SYSTEM	70
11.7	THE GENERAL PURPOSE GRAPHICS SYSTEM	71
11.8	SOME ADDITIONAL COMMENTS	71
	REFERENCES	74

SECTION 1: INTRODUCTION

This document contains a description of the internal operation of the SLAC Unified Graphics System. It includes descriptions of how the various modules in the system communicate with each other, and descriptions of all of the libraries and data sets which are necessary to the maintenance and development of the system. This document assumes that the reader is familiar with the Unified Graphics Programming Manual [BEA-1] and with Assembler Language programming for the IBM SYSTEM/360-370 computers.

The Unified Graphics System first became available at SLAC in December 1972. The original version of the Unified Graphics System is described in [BEA-2, BEA-3, BEA-4, BEA-5]. This document and [BEA-1] describes a second version of the system which became available in February 1976. The difference between the first and second versions is mostly internal, although a few subroutine calls were modified.

SECTION 1.1: A BRIEF DESCRIPTION OF THE LIBRARIES

There are a number of libraries and data sets related to the Unified Graphics System in addition to the ones used directly by a user's program. The largest of these libraries is WYL.CG.RCB.UGSOURCE which contains the source code for all of the modules in the system. Some of the modules are written in FORTRAN or PL/1, but most are written in IBM SYSTEM/360-370 Assembler Language.

Most of the modules coded in Assembler Language are written in a language-independent manner; the language-dependent segments of the modules are coded as macro calls in the module. When a FORTRAN object deck is to be generated from its Assembler Language source module, the source module is assembled with the FORTRAN macro library, WYL.CG.RCB.UGFTNMAC, made available to the Assembler. If a PL/1 Optimizing Compiler object deck is required, the only difference is that a different macro library, WYL.CG.RCB.UGPLOMAC, is made available to the Assembler. The library necessary to produce a PL/1 F-Level module is WYL.CG.RCB.UGPLFMAC. The language-dependent macros are flexible enough that the calling sequences do not have to be the same in different supported languages.

In addition to a macro library for each supported language, there is a macro library, WYL.CG.RCB.UGRUNMAC, for the run time modules, and a macro library, WYL.CG.RCB.UGCOMMAC, which contains common macros which may be used by any of the modules. This latter library must be supplied to all Assembler Language modules

when they are assembled.

SECTION 1.2: A BRIEF DESCRIPTION OF THE CONTROL BLOCKS

There are three types of control blocks that are created and manipulated by the modules in this system. These control blocks are:

1. The Main Communication Area (MCA).
2. The Device Dependent Area (DDA).
3. The Attention Queue Element (AQE).

The purpose and use of each of these control blocks is described in the following paragraphs.

A user's program contains exactly one MCA and it is part of the load module. The more important items in the MCA are:

1. A pointer to the DDA of the active graphic device.
2. A pointer to the DDA and the identification of all of the open graphic devices.
3. The entry points to a number of important modules in the user load module. These modules include the error processor, options scanner, and modules to unpack a graphic element.

There is exactly one DDA for each open device. A DDA is created by invoking the GETMAIN macro when UGOPEN is called and it is deleted by invoking the FREEMAIN macro when UGCLOS is called. While the device is open, it contains all of the information that the system has available about the device. The first part of the DDA is the same for all devices and includes:

1. The entry point to the device-dependent code.
2. The device and programmer scaling and scissoring limits (current and original).
3. The name of the device-dependent code and the name of the device-dependent close module.
4. A pointer to the list of AQE's. The pointer may be null.
5. An Event Control Block (ECB) which UGRATN will WAIT on and which will be POST'ed when an attention occurs.

The second part of the DDA depends entirely on the device and is different for each device.

AQE control blocks are only used with interactive devices. When an enabled attention occurs on an interactive device, a section of the device-dependent code is invoked. This code does a GETMAIN macro for an AQE and saves all of the information about the attention in the AQE. A pointer to the AQE is put into the DDA of the device. When UGRATN is called, the AQE is disconnected from the DDA and the information in the AQE is transferred to UGRATN's output area, and the AQE is deleted by invoking the FREEMAIN macro.

SECTION 1.3: A DESCRIPTION OF A PROGRAM IN EXECUTION

This section will follow the execution of a very simple program from beginning to end. For definiteness, suppose the program runs on an IBM 2250 and puts a single picture on the screen, enables the keyboard, and waits for an attention. When the attention is received, the program terminates. The operations as they occur are:

1. The program calls UGOPEN. UGOPEN scans the options list and finds the IBM2250 item. This item is matched with a table and UGOPEN determines the name of the device-dependent open module, UGIB501 in this case. UGOPEN then LOAD's UGIB501 and calls it. UGIB501 begins by re-scanning the options list to get the device-dependent items. UGIB501 determines the length of the DDA and invokes GETMAIN for the required space. The DDA is then initialized and the Data Control Block (DCB) for the IBM 2250 is opened. Finally, UGIB501 returns to UGOPEN and UGOPEN deletes UGIB501 from memory. UGOPEN saves the pointer to the DDA in the HCA. Then the name of the device-dependent module, UGIB502 in this case, is obtained from the DDA and this module is brought into memory with the LOAD macro. UGOPEN then transfers control to UGIB502 to allow it to finish any necessary initialization; in the case of the IBM 2250, this includes telling the Operating System where to transfer control when an attention is generated by the console operator. When UGIB502 returns, the open operation is complete and UGOPEN returns.
2. The program calls UGEINT, UGETXT, etc. to create a graphic element. The device-dependent code is not entered by these modules.
3. The program calls UGEPUT to transmit the element to the device. UGEPUT creates a standard argument list and transfers control to the device-dependent code, UGIB502, of the active DDA. UGIB502 uses the entry points in the HCA to call the element unpacking modules. When the element has been converted to device-dependent orders, it is written to the IBM 2250, and a record of where the display orders are in the IBM 2250 is saved in the DDA.
4. The program calls UGRATN to enable the keyboard. This causes UGIB502 to be entered in the same way that UGEPUT entered it. UGIB502 sets a flag in the DDA to indicate that the keyboard in now enabled.
5. The program calls UGRATN to wait for an attention. UGRATN waits on the ECB in the active DDA. The user program is now suspended in the middle of UGRATN.
6. The console operator generates a keyboard attention. The device-dependent code is entered by the Operating System at the address previously specified. An AQE is created and a pointer to it is put in the DDA. Figure 1.3.1 shows the status of memory at this time. Then the ECB is posted. This causes UGRATN to be re-activated. UGRATN moves the information in the AQE into ATCODE, IARRAY, and XARRAY; and then deletes the

AQE.

7. The program calls UGCLOS. UGCLOS transfers control to the device-dependent code, UGIB502, of the active DDA. UGIB502 deletes the attention entry point and returns. UGCLOS then gets rid of UGIB502 with a DELETE macro, and obtains the name of the device-dependent close module, UGIB503 in this case, from the DDA and LOAD's and calls this module. UGIB503 closes the DCB for the IBM 2250, deletes any AQE's attached to the DDA and finally deletes the DDA itself. UGIB503 then returns to UGCLOS, which deletes UGIB503 from memory and itself returns to the main program.

If a subroutine detects an error at execution time, it calls the error processor and supplys it with the level of the error, name of the subroutine, and index of the error. The error processor loads the error message module for the subroutine and prints a full error message. After the error message has been printed, the error message module is deleted from memory.

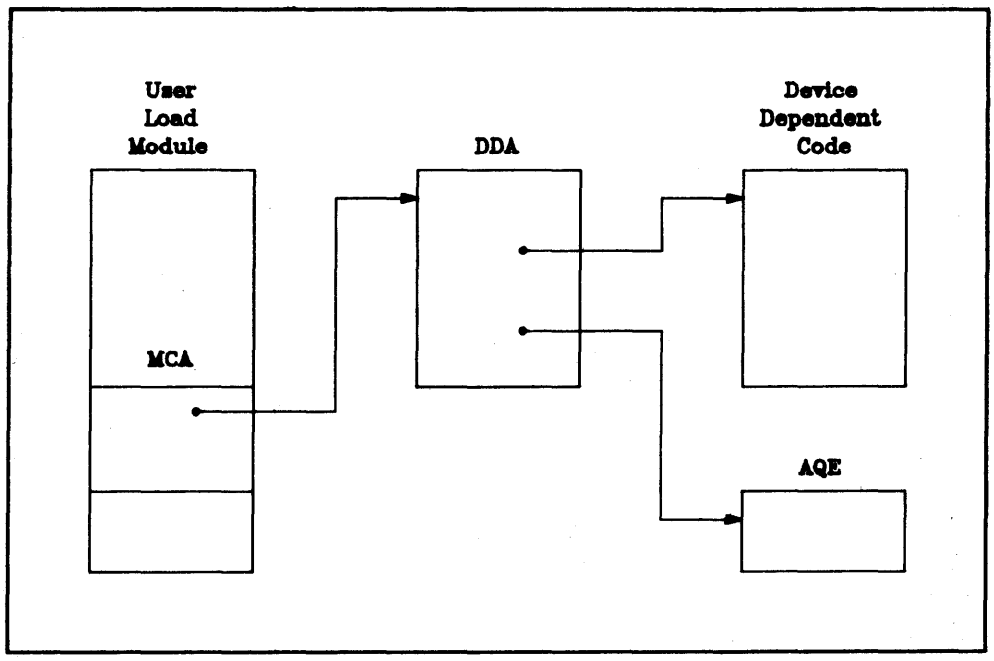

Figure 1.3.1: An Example of Memory Usage at Execution Time.

5 SECTION 2: A DETAILED DESCRIPTION OF THE CONTROL BLOCKS

This section gives a detailed description of the three types of control blocks that are created and manipulated by this system. These control blocks are accessible only to Assembler Language modules.

SECTION 2.1: THE MAIN COMMUNICATION AREA (MCA)

The MCA control block is part of the user's load module; its name in the LINK-EDITOR or LOADER map is UGX000. This module is the keystone of the Unified Graphics System. Any module that has access to the MCA is potentially able to determine anything about any open graphic device. The DSECT which defines the MCA is shown below:

```
          GBLC  &UGLAGID
          GBLA  &UGLAGDP
MCAX      DSECT                 MAIN COMMUNICATION AREA.
MCAIDENT  DC    CL4'MCA '         MCA IDENTIFICATION.
MCALNGTH  DC    A(MCANSIZE)       MCA LENGTH (IN BYTES).
MCALANGU  DC    CL4'&UGLAGID'     LANGUAGE IDENTIFICATION.
MCAPADDA  DC    F'0'              POINTER TO ACTIVE DDA.
MCANDEVC  EQU   8                 NUMBER OF CONCURRENT DEVICES.
MCADDAID  DC    (MCANDEVC)H'0'    DDA IDENTIFICATIONS.
MCADDAPR  DC    (MCANDEVC)F'0'    DDA POINTERS.
          SPACE 1
MCACHRID  DC    CL8'UGCHAR1 '     CHARACTER GEN. MODULE NAME.
MCACHREP  DC    A(0)              CHARACTER GEN. ENTRY POINT.
MCACHRLW  DC    A(MCACHRW1)       LENGTH OF CH. GEN. WORK SPACE.
MCACHRAW  DC    A(0)              ADDRESS OF CH. GEN. WORK SPACE.
MCACHRW1  EQU   768                 WORK SPACE SIZE FOR UGCHAR1.
MCACHRW2  EQU   1024                WORK SPACE SIZE FOR UGCHAR2.
MCACHRW3  EQU   1280                WORK SPACE SIZE FOR UGCHAR3.
MCACHRW4  EQU   2048                WORK SPACE SIZE FOR UGCHAR4.
MCACHRW5  EQU   2048                WORK SPACE SIZE FOR UGCHAR5.
MCACHRW6  EQU   2048                WORK SPACE SIZE FOR UGCHAR6.
MCACHRW7  EQU   4096                WORK SPACE SIZE FOR UGCHAR7.
MCACHRW8  EQU   8192                WORK SPACE SIZE FOR UGCHAR8.
          SPACE 1
MCAOELEM  DC    A(MCAOPTEL)       ADDR. OF OPTIONS FOR ELEMENTS.
          SPACE 1
MCAEPOSC  DC    V(UGX001)         ENTRY POINT...OPTIONS SCANNER.
MCAEPERR  DC    V(UGX002)         ENTRY POINT...ERROR PROCESSOR.
MCAEPPNT  DC    V(UGX101)         ENTRY POINT...POINT RETRIEVER.
MCAEPLIN  DC    V(UGX102)         ENTRY POINT...LINE RETRIEVER.
MCAEPSTR  DC    V(UGX103)         ENTRY POINT...STROKE RETRIEVER.
MCAEPCHR  DC    V(UGX104)         ENTRY POINT...CHAR. RETRIEVER.
MCAEPDTA  DC    V(UGX105)         ENTRY POINT...DATA RETRIEVER.
```

```
MCAEPCTX DC    V(UGX111)        ENTRY POINT...,COORD. TRANS.
MCAEPSCH DC    V(UGX112)        ENTRY POINT...,SCISSORING MOD.
MCAEPROT DC    V(UGX113)        ENTRY POINT...ROTATION CALC.
MCAEPBEG DC    V(UGX201)        ENTRY POINT...DSA ALLOCATION.
MCAEPEND DC    V(UGX202)        ENTRY POINT...DSA TERMINATION.
         SPACE 1
MCALAGLG DC    F'&UGLAGDP'      LENGTH OF LANGUAGE DEP. AREA.
MCALAGDP DC    (&UGLAGDP)F'0'   LANGUAGE DEPENDENT AREA.
         SPACE 1
MCAOPTEL DC    H'21'            OPTIONS FOR ELEMENTS.
         DC    H'1,4,2,1',CL4'VSHL'
         DC    H'1,4,2,2',CL4'SHAL'
         DC    H'1,4,2,4',CL4'LARG'
         DC    H'1,4,2,8',CL4'VLRG'
         DC    H'1,4,4,1',CL4'SOLD'
         DC    H'1,4,4,2',CL4'DASH'
         DC    H'1,4,4,4',CL4'DDSH'
         DC    H'1,4,4,8',CL4'DOTS'
         DC    H'1,4,6,1',CL4'VDIM'
         DC    H'1,4,6,2',CL4'DIMM'
         DC    H'1,4,6,4',CL4'BRIT'
         DC    H'1,4,6,8',CL4'VBRT'
         DC    H'1,4,8,1',CL4'COL1'
         DC    H'1,4,8,2',CL4'COL2'
         DC    H'1,4,8,4',CL4'COL3'
         DC    H'1,4,8,8',CL4'COL4'
         DC    H'1,4,10,1',CL4'WINK'
         DC    H'1,4,10,2',CL4'STDY'
         DC    H'3,8,12,0',CL8'XSPACING'
         DC    H'3,7,16,0',CL8'SPACING '
         DC    H'3,5,20,0',CL6'ANGLE '
         SPACE 1
MCAMSIZE EQU   *-MCAIDENT
```

The first two lines of code define language-dependent parameters. &UGLAGID is a language identification which receives the value 'FORT' for FORTRAN, 'PL10' for the PL/1 Optimizing Compiler, and 'PL1F' for the PL/1 F-Level Compiler. &UGLAGDP defines the length, in words, of a language-dependent area in the MCA. It has a value of 14 in FORTRAN and zero for both PL/1 compilers.

Within the DSECT itself, the first three items (MCAIDENT, MCALNGTH, and MCALANGU) contain control block identification and are never modified by the program in execution. Next in the MCA is MCAPADDA, a pointer to the active DDA, and arrays, MCADDAID and MCADDAPR, which contain the identifications of all of the open graphic devices and pointers to their DDA's. These pointers and identifications are manipulated by subroutines UGOPEN, UGCLOS, and UGSLCT.

The next group of entries in the MCA are concerned with the active character stroke generator. The entries are initialized as shown. The device-dependent code calls subroutine UGX103 when strokes must be generated for a character string. If MCACHREP is zero, UGX103 will load the character stroke generator into memory and save its entry point in MCACHREP. Also, if MCACHRAW is zero, UGX103 allocates the work space and saves its address in

MCACHRAM. Subroutine UGCSET also manipulates these entries. UGCSET can delete a character generator from memory and free the allocated work space. If UGCSET is changing the current character generator, the new work space requirement is smaller than the old requirement, and the DELETE options item is not given, then the old work space is not deleted. Thus, if a program is repeatedly switching between character generators, the work space will grow to its largest length and will not change after that.

The MCA contains equals cards for eight character stroke generators while the Unified Graphics System only contains three. The purpose of this is to make it easy to add temporary or experimental character stroke generators to the system without recompiling existing subroutines. To accomplish this, UGCSET contains five options items that are not documented in the Unified Graphics Programming Manual [BEA-1]. These items are CSET4, CSET5, CSET6, CSET7, and CSET8. The item CSET7, for example, will cause MCACHRID to be set to 'UGCHAR7' and MCACHRLN to be set to 4096. Thus, a temporary module with the name UGCHAR7 could be supplied and this module would have a work space of 4096 bytes. The Unified Graphics System contains dummy modules named UGCHAR4 through UGCHAR8 so that incorrect attempts to use an experimental stroke generator will not cause the program to ABEND.

MCAOPLEN contains the address of the options input structure for the graphic element generation subroutines. Putting this structure in the MCA means that it does not have to be repeated in subroutines UGEPNT, UGELIN, UGETXT, etc.

The next group of entries in the MCA are a number of entry points to modules within the user's load module that must be accessed by the dynamically loaded modules. The dynamically loaded modules always have a pointer to the MCA in their argument list and they are, therefore, able to call any of the listed modules.

After the entry points are some language-dependent entries. The array MCALAGDP is not needed in the PL/1 versions, but an array of 14 words is allocated in FORTRAN. The first word is used in the FORTRAN version of subroutine UGX002 to handle recursive use of the error processor. If the first byte of this word is X'00', then recursion is not permitted. The second byte is set to X'FF' when UGXERR is active. If the user supplies a FORTRAN compatible version of UGXERR which is capable of supporting recursive use, then the first thing that this subroutine should do is to set the first byte of MCALAGDP to a non-zero value. The remainder of the space is used in managing the space allocation for DSA's for each subroutine call. Starting in the third word are four blocks of three words each. The first word in each block is the address of the task control block (TCB) of the task to which the space is assigned, the second word is the address of the allocated block, and the third word is the address of the free space within the block. DSA's are allocated from this block unless it will not fit into the block, in which case a separate GETMAIN is issued for the DSA. Most user programs only contain a single task, but it is possible to have four tasks running at once provided the

user insures that I/O requests to a graphic device are properly scheduled. The first byte of the second word in the FORTRAN language-dependent area contains a flag which is used to deallocate the memory blocks. When UGCLOS detects that no more devices are open, it calls UGX203 which sets this flag byte to X'FF'. When the next call to UGX202 is made, which must necessarily come from UGCLOS, the blocks for all of the tasks will be deallocated.

The format of the MCA is defined by a macro in the macro library which must be supplied to all Assembler Language subroutines.

SECTION 2.2: THE DEVICE-DEPENDENT AREA (DDA)

Each graphic device which is open has a DDA control block associated with it. All of the necessary information about the device is contained in the DDA. The first part of a DDA is the same for all devices; the remainder is specialized for the device and is only used by the device-dependent modules for that device. A DDA control block is created and initialized by the device-dependent open module and is deleted by the device-dependent close module. The common part of the DDA is shown below:

```
DDAX     DSECT                  COMMON DDA SECTION.
DDAIDENT DS    CL4'DDA '          DDA IDENTIFICATION.
DDALNGTH DS    A                  DDA LENGTH (IN BYTES).
DDAPTMCA DS    V(UGX000)          POINTER TO MCA.
DDADEVID DS    H                  DEVICE IDENTIFICATION.
DDADEVTY DS    CL8                DEVICE TYPE.
         SPACE 1
DDADPRCH DS    CL1                DEV. PROP...CHARACTER SIZES.
DDADPRC1 EQU   X'01'                VSML.
DDADPRC2 EQU   X'02'                SMAL.
DDADPRC3 EQU   X'04'                LARG.
DDADPRC4 EQU   X'08'                VLRG.
DDADPRLS DS    CL1                DEV. PROP...LINE STRUCTURE.
DDADPRL1 EQU   X'01'                SOLD.
DDADPRL2 EQU   X'02'                DASH.
DDADPRL3 EQU   X'04'                DDSH.
DDADPRL4 EQU   X'08'                DOTS.
DDADPRIL DS    CL1                DEV. PROP...INTENSITY LEVEL.
DDADPRI1 EQU   X'01'                VDIM.
DDADPRI2 EQU   X'02'                DIMM.
DDADPRI3 EQU   X'04'                BRIT.
DDADPRI4 EQU   X'08'                VBRT.
DDADPRCR DS    CL1                DEV. PROP...COLOR.
DDADPRR1 EQU   X'01'                COL1.
DDADPRR2 EQU   X'02'                COL2.
DDADPRR3 EQU   X'04'                COL3.
DDADPRR4 EQU   X'08'                COL4.
DDADPRWM DS    CL1                DEV. PROP...WINK MODE.
DDADPRW1 EQU   X'01'                WINK.
DDADPRW2 EQU   X'02'                STDY.
```

```
DDADPREX DS     CL1             DEV. PROP...PICTURE EXTENSION.
DDADPRX1 EQU    X'01'             UP.
DDADPRX2 EQU    X'02'             RIGHT.
DDADPRX3 EQU    X'04'             DOWN.
DDADPRX4 EQU    X'08'             LEFT.
DDADPRHW DS     CL1             DEV. PROP...INTER. HARDWARE.
DDADPRH1 EQU    X'01'             KEYBOARD.
DDADPRH2 EQU    X'02'             LIGHT PEN.
DDADPRH3 EQU    X'04'             SCREEN POINTER.
DDADPRSC DS     CL1             DEV. PROP...SCREEN TYPE.
DDADPRS1 EQU    X'01'             NON-INTERACTIVE.
DDADPRS2 EQU    X'02'             INTERACTIVE/RE-FRESH.
DDADPRS3 EQU    X'04'             INTERACTIVE/STORAGE.
DDADPRXX DS     CL2             UNUSED BYTES.
DDADPRCZ DS     4F              DEV. PROP...CHAR. SIZE VALUES.
DDADPRSZ DS     1F              DEV. PROP...SCREEN SIZE VALUE.
         SPACE  1
DDAFLGSC DS     CL1             DEV. FLAG...SCISSORING.
DDAFLGS1 EQU    X'00'             SCISSORING REQUIRED.
DDAFLGS2 EQU    X'FF'             SKIP SCISSORING.
DDAFLGXX DS     CL3             UNUSED BYTES.
         SPACE  1
DDARSTBK DS     8H'0'           RUN STATISTICS...ACCUM. BLOCK.
DDARSTSZ EQU    *-DDARSTBK      RUN STATISTICS...BLOCK SIZE.
         SPACE  1
DDASCLDO DS     4F              ORIGINAL DEVICE SCALING LIMITS.
DDASCLPO DS     4E              ORIGINAL PROG. SCALING LIMITS.
DDASCLDC DS     4F              CURRENT DEVICE SCALING LIMITS.
DDASCLPC DS     4E              CURRENT PROG. SCALING LIMITS.
         SPACE  1
DDADDMOD DS     CL8             NAME OF DEVICE DEP. MODULE.
DDACLOSE DS     CL8             NAME OF CLOSE MODULE.
DDAEPDDC DS     V               ENTRY POINT TO D. D. CODE.
         SPACE  1
DDAATECB DS     F'0'            ATTENTION ECB.
DDAATPTR DS     F               POINTER TO NEXT QUEUED ATTN.
DDAATPT1 EQU    X'FF'             NON-INTERACTIVE DEVICE FLAG.
DDAATAFG DS     F               ASYNCHRONOUS ATTENTION FLAG.
DDAATAF1 EQU    X'FF'             ASYNC. ATTN. PRESENT.
```

The first part of the DDA contains identifying information about the control block. DDADEVID is the identification value supplied by UGOPEN and DDADEVTY is the options item from UGOPEN which selects the device.

The next group of items contain information about the graphic device. This information is all initialized in the device-dependent open module and is obtainable through subroutine UGDINF. The following group of items is controlled by UGSCIS and can be used to turn scissoring on and off. Next, comes entries containing the run time statistics which can be retrieved or reset by subroutine UGRINF. After these items, comes a group of entries containing the original and current device and programmer scaling and scissoring limits. These limits may be manipulated with subroutines UGRSCL and UGSCAL.

In the next group of items are the names and address of device-dependent modules. DDADDMOD and DDACLOSE contain the names of the device-dependent code and the device-dependent close module. These values are initialized by the device-dependent open module. UGOPEN uses DDADDMOD to load the device-dependent module and saves its entry point in DDAEPDDC. DDACLOSE is used by subroutine UGCLOS to load the device-dependent close module.

The next group of entries are really only utilized on interactive devices. DDAATECB is an Event Control Block (ECB) which is POST'ed by the device-dependent code when an attention is available. When an attention is available, DDAATPTR will point to the oldest AQE for the device. DDAATPTR is initialized to zero for all devices, and for non-interactive devices, the first byte is set to DDAATPT1. DDAATAFG is used to record the occurrence of an asynchronous error. These are errors which cannot be reported to the error processor when they occur because, for example, the error processor may be busy with something else. Such an error can occur, for example, when a parity check occurs in the IBM 2250 Display Buffer. These errors are processed in the following manner: (1) the device-dependent code sets the first byte of DDAATAFG to DDAATAF1 and puts an error code in the two low order bytes of DDAATAFG. (2) A special AQE is generated containing the error code and the name of the device-dependent code. (3) If UGRATN finds one of these special AQE's, it reports the error as a level four problem and clears DDAATAFG. If DDAATAFG is already clear when UGRATN finds the AQE, it ignores the AQE. (4) The device-dependent code always checks DDAATAFG when it is entered and reports any error and clears DDAATAFG. In this manner, these asynchronous problems are reported to the error processor at the first possible moment.

Each individual device has a different extension to the basic DDA described above. The extension contains such things as pointers to the DCB, pointers to the output buffers or the buffers themselves, current pen position, or any of the other items that must be retained from one call to a Unified Graphics subroutine to the next. The extensions to the basic DDA are defined in the macro library for the run time subroutines. The common part of the DDA is defined in a macro in the macro library which must be supplied to all subroutines.

SECTION 2.3: THE ATTENTION QUEUE ELEMENT (AQE)

AQE control blocks are created by the device-dependent code when an attention must be reported to the user's program. The AQE is deleted by UGRATN when the information in the AQE has been transmitted to the user's program. The DSECT, which defines an AQE, is shown below:

```
AQEX     DSECT                ATTENTION QUEUE ELEMENT.
AQEIDENT DS    CL4'AQE '        AQE IDENTIFICATION.
AQELNGTH DS    A(AQESIZE)       AQE LENGTH (IN BYTES).
```

```
AQEPTAQE DS    F              POINTER TO NEXT AQE.
AQEUTIME DS    F              UNEXPIRED TIME.
AQEACODE DS    CL4            ATTENTION CODE.
AQEADATA DS    8F             ATTENTION DATA.
         DS    0D
AQESIZE  EQU   *-AQEX         LENGTH OF AQE.
```

As in all of these control blocks, the first two entries are an identification field and a length.

The next entry, AQEPTAQE, is used to chain AQE's together. The DDAATPTR field of the DDA points to the oldest AQE that has been created but not deleted. If more than one AQE exists, the AQEPTAQE field points to the next oldest one. A zero in this field indicates the end of the chain.

The next three entries, AQEUTIME, AQEACODE, and AQEADATA contain information which is returned by UGRATN. When UGRATN reports a LPEN or SPTR attention, it converts the device-dependent coordinates in AQEADATA to programmer coordinates and puts these values into the floating point array in UGRATN's argument list.

The format of an AQE is defined by a macro in the macro library which must be supplied to all subroutines.

This section gives a description of all of the data sets and libraries that were used in the development and are used in the maintenance of the Unified Graphic System. One thing that all of the following source libraries have in common is a member named LIBLIST. This member consists of a job which may be submitted to produce a listing of the library.

At the time that this document is being released, all of these data sets and libraries, except as noted, are catalogued and are on the WYLBUR disk packs. It may be necessary in the future to move some of them to mountable disk packs or backup tapes.

SECTION 3.1: SOURCE DATA SETS

The data sets described below contain the source data necessary to create the object modules in the Unified Graphics System.

SECTION 3.1.1: THE LIBRARY WYL.CG.RCB.UGSOURCE

This source library is a WYLBUR format partitioned data set which contains the basic source code for all of the Unified Graphics modules.

Members of this library with six character names, UGOPEN for example, contain source code for Assembler Language modules and are coded in a language-independent manner. When these modules are assembled for one of the LINK-EDITOR/LOADER libraries, a macro-library defining the language-dependent features must be made available to the Assembler.

Members of this library with eight character names are also destined for the LINK-EDITOR/LOADER libraries. If the name ends in "FT", the module is the FORTRAN version; if the name ends in "PL", the module is the PL/1 Optimizer and F-Level Compiler version. A name ending in "PO" is for the PL/1 Optimizing Compiler version only while a name ending in "PF" means that the module is the PL/1 F-Level version. The library, therefore, contains members with names like UGAXISFT, UGAXISPL, UGX002PO, and UGX002PF. Most of the modules with eight character names are coded in a higher-level language, but some of them are coded in Assembler Language. Some of the modules that are coded in FORTRAN will not compile under the FORTRAN-G Compiler; they require FORTRAN-H.

Still other members in this library have seven character names. These modules are destined for the run time libraries and are mostly coded in Assembler Language. The modules whose seventh character is an "X" are error message modules.

SECTION 3.1.2: THE LIBRARY WYL.CG.RCB.UGCOMMAC

This source library is a card image partitioned data set which contains macros which must be supplied to the Assembler whenever any module is being assembled. This library does not usually reside on disk; it is created when needed from the data set named WYL.CG.RCB.UGCOMCNL.

This library contains the definition of the language-independent part of the HCA, the common part of the DDA's, the AQE, and a group of macros which are used throughout the Assembler Language code.

SECTION 3.1.3: THE LIBRARY WYL.CG.RCB.UGFTNMAC

This source library is a card image partitioned data set which contains macros which must be supplied to the Assembler whenever a module is being assembled for the library WYL.CG.RCB.UGFTNLIB. This library does not usually reside on disk; it is created when needed from the data set named WYL.CG.RCB.UGFTNCNL.

This library contains the prologue and epilogue macros as well as the language-dependent parts of all of the FORTRAN modules coded in Assembler Language.

SECTION 3.1.4: THE LIBRARY WYL.CG.RCB.UGPLOMAC

This source library is a card image partitioned data set which contains macros which must be supplied to the Assembler whenever a module is being assembled for the library WYL.CG.RCB.UGPLOLIB. This library does not usually reside on disk; it is created when needed from the data set named WYL.CG.RCB.UGPLOCNL.

This library contains the prologue and epilogue macros as well as the language-dependent parts of all of the PL/1 Optimizing Compiler modules coded in Assembler Language.

SECTION 3.1.5: THE LIBRARY WYL.CG.RCB.UGPLFMAC

This source library is a card image partitioned data set which contains macros which must be supplied to the Assembler whenever a module is being assembled for the library WYL.CG.RCB.UGPLFLIB. This library does not usually reside on disk; it is created when needed from the data set named WYL.CG.RCB.UGPLFCNL.

This library contains the prologue and epilogue macros as well as the language-dependent parts of all of the PL/1 F-Level Compiler modules coded in Assembler Language.

SECTION 3.1.6: THE LIBRARY WYL.CG.RCB.UGRUNMAC

This source library is a card image partitioned data set which contains macros which must be supplied to the Assembler whenever a module is being assembled for the library WYL.CG.RCB.UGRUNLIB. This library does not usually reside on disk; it is created when needed from the data set named WYL.CG.RCB.UGRUNCNL.

This library contains macros to define the device-dependent extensions of the DDA's, the prologue and epilogue macros for the run time modules, as well as other macros which have been useful in developing the run time modules.

SECTION 3.1.7: THE DATA SET WYL.CG.RCB.UGCOMCNL

This source data set is a WYLBUR format sequential data set which is used to create the data set WYL.CG.RCB.UGCOMMAC. The data set consists of the members of WYL.CG.RCB.UGCOMMAC separated by control cards for the IEBUPDTE utility program. The suffix "CNL" in the data set name stands for Condensed Macro Library.

SECTION 3.1.8: THE DATA SET WYL.CG.RCB.UGFTMCNL

This source data set is a WYLBUR format sequential data set which is used to create the data set WYL.CG.RCB.UGFTMMAC. The data set consists of the members of WYL.CG.RCB.UGFTMMAC separated by control cards for the IEBUPDTE utility program.

SECTION 3.1.9: THE DATA SET WYL.CG.RCB.UGPLOCNL

This source data set is a WYLBUR format sequential data set which is used to create the data set WYL.CG.RCB.UGPLOMAC. The data set consists of the members of WYL.CG.RCB.UGPLOMAC separated by control cards for the IEBUPDTE utility program.

SECTION 3.1.10: THE DATA SET WYL.CG.RCB.UGPLFCNL

This source data set is a WYLBUR format sequential data set which is used to create the data set WYL.CG.RCB.UGPLFMAC. The data set consists of the members of WYL.CG.RCB.UGPLFMAC separated by control cards for the IEBUPDTE utility program.

SECTION 3.1.11 THE DATA SET WYL.CG.RCB.UGRUNCHL

This source data set is a WYLBUR format sequential data set which is used to create the data set WYL.CG.RCB.UGRUNMAC. The data set consists of the members of WYL.CG.RCB.UGRUNMAC separated by control cards for the IEBUPDTE utility program.

SECTION 3.1.12: THE LIBRARY WYL.CG.RCB.UGPL1DCL

This source library is a card image partitioned data set containing the entry declarations for the PL/1 subroutines which a user's program may call. A user may make this library available to the PL/1 Compiler by means of the SYSLIB data set. Most of the PL/1 source modules in WYL.CG.RCB.UGSOURCE require this library when they are compiled.

SECTION 3.2: OBJECT DATA SETS

The data sets described below contain all of the object modules which are necessary when using the Unified Graphics System.

SECTION 3.2.1: THE LIBRARY WYL.CG.RCB.UGFTNLIB

This object library must be made available to the LINK-EDITOR or LOADER by means of the SYSLIB data set whenever these utilities are processing programs using the FORTRAN version of the Unified Graphics System. The LINK-EDITOR or LOADER will select those members from this data set which have been called by the user's program. The modules in this library do not contain any known SLAC system-dependencies.

SECTION 3.2.2: THE LIBRARY WYL.CG.RCB.UGPLOLIB

This object library must be made available to the LINK-EDITOR or LOADER by means of the SYSLIB data set whenever these utilities are processing programs using the PL/1 Optimizing Compiler version of the Unified Graphics System. The LINK-EDITOR or LOADER will select those members from this data set which have been called by the user's program. The modules in this library do not contain any known SLAC system-dependencies.

SECTION 3.2.3: THE LIBRARY WYL.CG.RCB.UGPLFLIB

This object library must be made available to the LINK-EDITOR or LOADER by means of the SYSLIB data set whenever these utilities

are processing programs using the PL/1 F-Level Compiler version of the Unified Graphics System. The LINK-EDITOR or LOADER will select those members from this data set which have been called by the user's program. The modules in this library do not contain any known SLAC system-dependencies.

SECTION 3.2.4: THE LIBRARY WYL.CG.RCB.UGRUNLIB

This object library must be made available at execution time by means of the STEPLIB or JOBLIB data set. At execution time, various modules will be loaded from this library when they are needed. This library contains utility modules, the character stroke generators, and the various device-dependent modules.

Some of the modules in this library are obviously SLAC-dependent. Examples are the device-dependent code which communicates through the SLAC Real-Time Network [DEN-1, DEN-2, GRA-1, DEN-3], or the device-dependent code for the interactive use of the TEKTRONIX 4013 which uses a group of HILTEN interface subroutines. Other non-obvious examples are the utility modules UGUTIL1 and UGUTIL2 which are described in a later section.

SECTION 3.3: OTHER DATA SETS

The data sets described below contain miscellaneous items used in the development and maintenance of the Unified Graphics System.

SECTION 3.3.1: THE DATA SET WYL.CG.RCB.UGPGMDOC

This data set is a WYLBUR format sequential data set which contains a job which produces the textual part of the Unified Graphics Programming Manual. The data set contains the complete JCL and all of the necessary control and data cards required by the FORMAT Text Processing Program. FORMAT is coded in FORTRAN and is available from:

SHARE Program Library Agency
Triangle Universities Computation Center
Post Office Box 12076
Research Triangle Park, North Carolina 27709
Telephone: (919) 549-0671 (Ext. 283)

as Distribution No. 360D-06.0.007. A short description of FORMAT is available in [BER-1] while a complete descripion is given in [EHR-1].

The figures for this document are produced by the program contained in the member UGPGMDOC in the library WYL.CG.RCB.UGTESTPG.

SECTION 3.3.2: THE DATA SET WYL.CG.RCB.UGINTDOC

This data set is a WYLBUR format sequential data set which contains a job which produces the textual part of the Unified Graphics Internal Operation and Maintenance Manual. The data set contains the complete JCL and all of the necessary control and data cards required by the FORMAT Text Processing Program.

The figures for this document are produced by the program contained in the member UGINTDOC in the library WYL.CG.RCB.UGTESTPG.

SECTION 3.3.3: THE LIBRARY WYL.CG.RCB.UGTESTPG

This source library is a WYLBUR format partitioned data set which contains a number of support programs and test programs for the Unified Graphics System. The members contain the complete JCL and all of the necessary source and data cards.

Some of the more important members in this library are:

1. UGPGMDOC This member is a job which will produce all of the figures in the Unified Graphics Programming Manual. The figures were drawn on the 10 inch CALCOMP Drum Plotter and these drawings were then photographically reduced to half their original size.
2. UGINTDOC This member is a job which will produce all of the figures in the Unified Graphics Internal Operation and Maintenance Manual.
3. UGMIDDPO A very extensive program which has been used to test the device-dependent code for all of the non-interactive devices in the Unified Graphics System.
4. UGIMDDPO A very extensive program which has been used to test the device-dependent code for all of the interactive devices in the Unified Graphics System.
5. UGSTARTX An interface module that allows modules compiled under the PL/1 Optimizing Compiler, like UGIMDDPO, to run under the SLAC Real-Time Network [DEM-1, DEM-2, GRA-1, DEM-3].
6. UGCHTEST An interactive program which can be used to give the character stroke generator a complete check-out.
7. EXAMPLE1 The example from Section 6.1 in the Unified Graphics Programming Manual.
8. EXAMPLE2 The example from Section 6.2 in the Unified Graphics Programming Manual.
9. EXAMPLE3 The example from Section 6.3 in the Unified Graphics Programming Manual.
10. EXAMPLE4 The example from Section 6.4 in the Unified Graphics Programming Manual.
11. SUMSUMRY A program which will analyze and summarize the SMF records produced by the utility module UGUTIL2.

The programs in UGPGMDOC, UGINTDOC, UGMIDDPO, UGIMDDPO, and SMFSUMRY are all written for the PL/1 Optimizing Compiler. These programs can all be converted to run under the PL/1 F-Level Compiler by adding entry declarations for the internal

procedures.

SECTION 3.3.4: THE LIBRARY WYL.CG.RCB.UGRUNMOD

This object library contains compiled versions of some of the programs in WYL.CG.RCB.UGTESTPG. For example, compiled versions of UGMIDDPO and UGINDDPO are in this library.

SECTION 3.3.5: THE LIBRARY WYL.CG.RCB.UGEXFILE

This source library is a WYLBUR format partitioned data set which contains a group of WYLBUR EXEC files which have been used in the generation and maintenance of the Unified Graphics System. These EXEC files and their uses are:

1. INVENTRY This EXEC file prints a complete inventory of all of the Unified Graphics data sets and the names of the members in the libraries.
2. CMLTOMAC This EXEC file will submit a job which generates a macro library from a condensed macro library. The reason for maintaining the libraries in the condensed form is that the condensed form takes only about 40% as much space as the expanded form.
3. SCRATMAC This EXEC file will scratch a macro library from a disk pack. With so many data sets with similar names, it is easy to make a mistake and accidentally scratch the wrong data set; this EXEC file checks the name carefully before allowing the data set to be deleted.
4. SRCTOLIB This EXEC file obtains a member from WYL.CG.RCB.UGSOURCE, adds JCL to assemble or compile the module and LINK-EDIT it into the proper object library, and submits the job
5. TAPEGENR This EXEC file submits a job which will write many of the Unified Graphics data sets onto a backup or distribution tape. The source data sets are written onto the tape by the WYLBUR utilities WUNPRESS and WLIBLIST. The object data sets are unloaded by either IEHMOVE or IEBCOPY, depending on the parameters typed while the TAPEGENR EXEC file is running.
6. TAPEVERF This EXEC file submits a job which will verify a tape prepared by TAPEGENR. The job will print the source files with IEBPTPCH and load the object files with either IEHMOVE or IEBCOPY.

SECTION 3.4: FORMAT OF THE BACKUP/DISTRIBUTION TAPE

This section describes the format of the tape prepared by the TAPEGENR EXEC file. The tape contains sixteen files, eleven files contain source data and five files contain unloaded object

modules. The five files containing object modules are unloaded from a 2314 disk pack. All source data sets have a RECFM of FB, an LRECL of 80, and a BLKSIZE of 4000. The DSNAME's of the files on the tape and their contents are:

1. UGPGMDOC The source for the Unified Graphics Programming Manual.
2. UGINTDOC The source for the Unified Graphics Internal Operation and Maintenance Manual.
3. UGSOURCE The source modules for the Unified Graphics system. The file consists of the members of WYL.CG.RCB.UGSOURCE separated by control cards for IEBUPDTE.
4. UGCOMMAC The common macro library with the members separated by IEBUPDTE control cards.
5. UGFTNMAC The macros for the FORTRAN version with the members separated by IEBUPDTE control cards.
6. UGPLOMAC The macros for the PL/1 Optimizing Compiler version with the members separated by IEBUPDTE control cards.
7. UGPLFMAC The macros for the PL/1 F-Level Compiler version with the members separated by IEBUPDTE control cards.
8. UGRUNMAC The macros for the run time modules with the members separated by IEBUPDTE control cards.
9. UGPL1DCL The entry declarations for the PL/1 callable modules. The members are separated by IEBUPDTE control cards.
10. UGTESTPG Source modules for support and test programs separated by IEBUPDTE control cards.
11. UGEXFILE Source for the WYLBUR EXEC files separated by IEBUPDTE control cards.
12. UGFTNLIB An unloaded object library containing the FORTRAN modules.
13. UGPLOLIB An unloaded object library containing the PL/1 Optimizing Compiler modules.
14. UGPLFLIB An unloaded object library containing the PL/1 F-Level Compiler modules.
15. UGRUNLIB An unloaded object library containing the run time modules.
16. UGRUNMOD An unloaded object library containing some test programs.

The unloaded object libraries, that is, the last five data sets, may have been generated by either IEHMOVE or IEBCOPY depending on information supplied to the TAPEGENR EXEC file.

This section describes, in detail, the content of a graphic element. A programmer normally constructs graphic elements by calling the procedures UGELIN, UGETXT, etc. It is possible for a programmer to construct graphic elements directly from the information given here. However, it should be remembered that little checking for invalid graphic elements is done at execution time.

A graphic element is an array of full words reserved to contain picture description information. Basically, the content of a graphic element is a few items of information at the beginning, followed by "blocks" of picture data. Each block begins with a "block specification" and contains either points, lines, text, or other miscellaneous information. In detail, this is:

```
  ELEMENT(1):  Current length of the graphic element (N).
  ELEMENT(2):  The index of the start of the first block (J).
  ELEMENT(3):  The  index of the start of the last block (K).
     ...             ...
  ELEMENT(J):  The start of the first block.
     ...             ...
  ELEMENT(K):  The start of the last block.
     ...             ...
  ELEMENT(N+1): The maximum length of the graphic element,
                that is, the dimension of the array minus one.
```

Notice that if a graphic element is to be saved in a data set, ELEMENT(1) gives the number of words which must be written. The value in ELEMENT(N+1) need not be saved because its value will depend on the size of the array that contains the graphic element when it is retrieved.

When procedure UGEINT is called with the CLEAR option, it resets the graphic element to the following:

```
  ELEMENT(1):  3
  ELEMENT(2):  4
  ELEMENT(3):  0
  ELEMENT(4):  HBOUND(ELEMENT,1)-1
```

where HBOUND is the PL/1 built-in function or its equivalent. When the RESET option is used, UGEINT sets:

```
  ELEMENT(ELEMENT(1)+1):  HBOUND(ELEMENT,1)-1
```

The CONTINUE option instructs UGEINT to do the same thing as for CLEAR unless the last block specification was for line data. In that case it sets:

```
  ELEMENT(1):  N
  ELEMENT(2):  4
  ELEMENT(3):  4
  ELEMENT(4)...ELEMENT(N):  A copy of the block specification
               for  the lines and the last end point from the
               line  data  with the blanking bit set to '0'B.
  ELEMENT(N+1):  HBOUND(ELEMENT,1)-1
```

The graphic element generators work by moving the word with the maximum length down, and inserting a block of data into the element. ELEMENT(1), and if necessary ELEMENT(3), are then reset. The graphic element generators use the pointer to the last block to avoid adding a new block specification when it is not needed.

SECTION 4.1: POINT DATA

A point data block consists of the following:

ELEMENT(I): The first character contains the character "P" and the second halfword contains the number of full words in the block.

ELEMENT(I+1): The intensity, color, and wink bits. Bits 12 through 15 are reserved for VBRT, BRIT, DIMM, and VDIM; bits 20 through 23 are for COL4, COL3, COL2, and COL1; and bits 30 and 31 are reserved for STDY and WINK.

Following this are a group of X and Y coordinates in floating point form.

SECTION 4.2: LINE DATA

A line data block consists of the following:

ELEMENT(I): The first character contains the character "L" and the second halfword contains the number of full words in the block.

ELEMENT(I+1): The line structure, intensity, color, and wink bits. Bits 4 through 7 are reserved for DOTS, DDSH, DASH, and SOLD.

Following this are a group of X and Y coordinates, in floating point form, of the end points of the line segments. The blanking bit is in bit 31 of the Y coordinate.

SECTION 4.3: TEXT DATA

A text data block consists of the following:

ELEMENT(I): The first character contains "C"; the second contains "1" (for text defined with the SPACING option), "2" (for XSPACING text), or "3" (for VSML, SMAL, LARG, or VLRG text); and the second halfword contains the number of full words in the block.

ELEMENT(I+1): The intensity, color, and wink bits.

ELEMENT(I+2): Either the character spacing for the stroke

generator in floating point form or the character sizes VLRG, LARG, SMAL, and VSML coded in bits 28 through 31.

ELEMENT(I+3): The rotation angle in floating point form.

Following this may be an arbitrary number of sub-blocks consisting of the following:

ELEMENT(I): X coordinate of the center of the first character in floating point form.

ELEMENT(I+1): Y coordinate of the center of the first character in floating point form.

ELEMENT(I+2): The start of the characters, four to a word and one to a byte, with the first byte containing the number of characters.

SECTION 4.4: DEVICE-DEPENDENT DATA

A data block containing information which is to be transmitted to the display device in unmodified form consists of the following:

ELEMENT(I): The first character contains the character "X" and the second halfword contains the number of full words in the block.

ELEMENT(I+1): The line structure, intensity, color, and wink bits.

Following this may be an arbitrary number of sub-blocks consisting of the following.

ELEMENT(I): X coordinate associated with the data in floating point form.

ELEMENT(I+1): Y coordinate associated with the data in floating point form.

ELEMENT(I+2): The start of the information, four bytes to the word, with the first byte containing the number of bytes in the sub-block.

The following sections describe the basic structure of the Assembler Language modules. These sections serve as an introduction to the Assembler Language listings for the Unified Graphics System.

Certain conventions are obeyed by essentially all of the Assembler Language modules in this system. The principal convention is in register usage. Only four general registers are ever used directly and these are identified symbolically as A0, A1, B0, and B1. The A registers constitute an even-odd pair as do the B registers. The floating point registers are limited to F0, and F1. All other register usage, including base registers, is handled internally by macros.

SECTION 5.1: MODULES INDEPENDENT OF THE DEVICE-DEPENDENT CODE

This section describes the structure of those modules which do not enter the device-dependent code. UGEINT is one of these modules and it is described below. Other modules of this nature are UGEPNT, UGELIN, UGETXT, UGEPTS, UGELNS, UGSLCT, UGDINF, UGRINF, UGODEV, UGSCAL, UGSCIS, UGCSET, UGCHAR, UGCTOL, UGRERR, UGOPTN, and UGX001.

```
EINT     TITLE 'UGEINT...INITIALIZE A GRAPHIC ELEMENT'
         PRINT ON,GEN,DATA
         GBLA  &UGNWKSP
         SPACE 4
UGEINT   UGPRLOG NCA=YES        INITIALIZE PROGRAM.
         UGEINT1                LANGUAGE DEP. PROCESSING.
    ...  Main Section of Code  ...
         UGEPLOG                RETURN TO CALLER.
         SPACE 2
         LTORG                  PROGRAM LITERALS.
         SPACE 4
         UGMCA                MAIN COMMUNICATION AREA.
         SPACE 4
DSAX     DSECT                DYNAMIC STORAGE AREA.
SAVE     DS    40F              SAVE AREA.
ARGL     DS    4F               PROCESSED ARGUMENT LIST.
ARGO     EQU   ARGL               OPTIONS LIST.
ARGA     EQU   ARGL+4             ADDRESS OF ELEMENT.
ARGM     EQU   ARGL+8             MULTIPLIER OF ELEMENT.
ARGN     EQU   ARGL+12            LENGTH OF ELEMENT.
    ...  Other Variables in DSA  ...
WKSP     DS    (&UGNWKSP)F      LANGUAGE DEP. WORK SPACE.
         DS    0D
NDSA     EQU   *-DSAX           LENGTH OF DSA.
```

```
SPACE 4
END
```

The first item of interest in this module skeleton is the third line. The global macro variable UGNWKSP defines the length of a language-dependent work space. More will be said about this variable in the following paragraphs.

The next interesting line is the UGPRLOG macro. This macro expands into a prologue for the module; it is defined in the macro library for each supported programming language. Some of the things that this macro does is: (1) save the current registers in the save area provided by the calling module, (2) obtain a Dynamic Storage Area (DSA) for the module, (3) link the save areas together, (4) establish a base register for the program, and (5) establish a base register for the MCA and/or DDA. Subroutine UGEINT only accesses the MCA and not the DDA, so the only macro parameter is MCA=YES. If this subroutine also needed to access the DDA, a parameter of DDA=YES would also be given. The PL/1 versions use the standard PL/1 schemes for allocating DSA's while the FORTRAN version calls a subroutine named UGX201 to do this function.

The next line invokes the UGEINT1 macro which is also defined in the macro library for each supported programming language. This macro takes the argument list, as supplied by the calling module, and generates a processed argument list in the DSA. The calling module may supply different calling sequences for each supported language but the processed argument list is the same for all programming languages. The macro UGEINT1 also assigns a value to the global macro variable UGNWKSP.

Following the main section of code for UGEINT is an invocation of the UGEPLOG macro. This macro expands into an epilogue for the module. Some of the things that this macro does is: (1) free the space allocated to the DSA, and (2) return to the calling module.

Some modules have a second macro for language-dependent processing immediately before the epilogue macro. If UGEINT had one, it would be called UGEINT2 and would be invoked immediately before the UGEPLOG macro. These second macro calls are in all modules which have output variables; the macro matches the computed values with the data types required by the supported language. Generally, it is these two macros which make use of the language-dependent work space in the DSA.

The next thing of interest in the module is the UGMCA macro. This macro expands into a DSECT for the complete MCA. If this module also used the DDA, it would invoke the UGDDA macro. The UGDDA macro expands into a DSECT for the common part of all DDA's.

The final items in the module skeleton constitute the definition of the DSECT for the DSA. The items shown in the DSA, are (1) the save area, which must come first in the DSA, (2) the processed argument list, and (3) the language-dependent work space. Notice that the data about the array ELEMENT in the

processed argument list contains something called a "multiplier". This multiplier is the byte offset between consecutive entries in the array. The multiplier always has a value of 4 in FORTRAN, but may take on other values in PL/1.

SECTION 5.2: MODULES WHICH ENTER THE DEVICE-DEPENDENT CODE

In this section are described those modules which enter the device-dependent code. Only a limited number of modules may enter the device-dependent code. For non-interactive devices, these modules are: UGOPEN, UGCLOS, UGEPUT, UGPICT, UGCTRL, and UGEGET. For interactive devices, the device-dependent code may also be entered by UGEATN, UGDATN, UGRATN, UGKPUT, and UGKGET. Certain device-dependent modules will also accept a call from subroutine UGETRN. The device-dependent code will not accept a call from any module except those listed here.

```
OPEN     TITLE 'UGOPEN...OPEN A GRAPHIC DEVICE'
         PRINT ON,GEN,DATA
         GBLA  &UGNWKSP
         SPACE 4
UGOPEN   UGPRLOG MCA=YES       INITIALIZE PROGRAM.
         UGOPEN1               LANGUAGE DEP. PROCESSING.
    ...  Main Section of Code  ...
         UGEPLOG               RETURN TO CALLER.
         SPACE 2
         LTORG                 PROGRAM LITERALS.
         SPACE 4
         UGMCA               MAIN COMMUNICATION AREA.
         SPACE 4
         UGDDA               DEVICE DEPENDENT AREA.
         SPACE 4
DSAX     DSECT               DYNAMIC STORAGE AREA.
SAVE     DS    40F             SAVE AREA.
ARGL     DS    9F              PROCESSED ARGUMENT LIST.
ARGN     EQU   ARGL              ARGUMENT LIST LENGTH.
ARGU     EQU   ARGL+4            NOT USED.
ARGX     EQU   ARGL+8            ADDRESS OF MCA.
ARGY     EQU   ARGL+12           MODULE INDEX.
ARGZ     EQU   ARGL+16           MODULE NAME.
ARGO     EQU   ARGL+24           OPTIONS LIST.
ARGI     EQU   ARGL+28           DEVICE IDENTIFICATION.
ARGD     EQU   ARGL+32           ADDRESS OF NEW DDA.
ARGV     EQU   *-ARGL            ARGUMENT LIST LENGTH.
    ...  Other Variables in DSA  ...
WKSP     DS    (&UGNWKSP)F     LANGUAGE DEP. WORK SPACE.
         DS    0D
NDSA     EQU   *-DSAX          LENGTH OF DSA.
         SPACE 4
         END
```

The principal difference between the skeleton for this module and the previous module is in the processed argument list. Certain

additions have been made to the beginning of this list because it is this list which is passed to the device-dependent code and the device-dependent code needs additional information, in particular, it needs the address of the HCA. The first item in the processed argument list is the length in bytes of the argument list. The second item is not used and the third item is a pointer to the HCA. The fourth item is an index that the device-dependent code uses to determine the identity of the module calling it, and the fifth item is the name of the calling module which is passed to the error processor if an error is detected. The actual processed arguments start with the sixth item in the list.

SECTION 5.3: INTERNAL MODULES

This section describes the structure of a group of modules which are called internally by the Unified Graphics System. The modules of this type are UGX101, UGX102, UGX103, UGX104, UGX105, UGX111, UGX112, UGX113, and UGX121.

```
X101     TITLE 'UGX101...RETRIEVE POINT DATA FROM AN ELEMENT'
         PRINT ON,GEN,DATA
         SPACE 4
UGX101   UGPRLOG NEWDSA=NO,DDA=YES
   ...   Main Section of Code  ...
         UGEPLOG NEWDSA=NO     RETURN TO CALLER.
         SPACE 2
         LTORG                   PROGRAM LITERALS.
         SPACE 4
ARGX     DSECT                 ARGUMENT LIST.
ARGY     DS    1F                NOT USED.
ARGW     DS    1F                ADDRESS OF WORK AREA.
ARGZ     DS    1F                ADDRESS OF DDA.
ARG1     DS    1C                INPUT FLAG.
ARG2     DS    1C                RESULT FLAG.
ARG3     DS    1C                SCISSORING FLAG.
ARG4     DS    1C                NOT USED.
ARGC     DS    2F                X,Y COORDINATES.
ARGA     DS    1F                ADDRESS OF POINT HEADING.
ARGM     DS    1F                MULTIPLIER OF POINT DATA.
         SPACE 4
         UGDDA                 DEVICE DEPENDENT AREA.
         SPACE 4
WKAX     DSECT                 WORK AREA.
SAVE     DS    16F               REGISTER SAVE AREA.
   ...   Other Variables in DSA  ...
WKA      EQU   *-WKAX            LENGTH OF WORK SPACE.
         SPACE 4
         END
```

The structure of these modules is somewhat different. Notice, first of all, the NEWDSA=NO argument in the UGPRLOG and UGEPLOG macros. This argument indicates that the macros are not to

allocate a fresh DSA but are to use a work area supplied by the calling program instead. The purpose of this is to minimize execution time in modules which may be called repeatedly. This module, UGX101, for example, is usually called once by the UGEPUT device-dependent code for each point found in a graphic element.

The argument list is described in a DSECT named ARGX. The first three words in the argument lists are the same for all of the modules in this class. The first word is not used, the second word is the address of the work area, and the third word is the address of the DDA. Following these three words are the arguments for the specific module.

Another thing to notice in this class of module is that the first item in the DSECT for the DSA is only 16 words long instead of the usual 40 words. The 40 word block is necessary in the other modules because some languages like PL/1 reserve part of the DSA after the normal register save area for special use. Since this class of module is only entered internally, this extra space is not necessary.

SECTION 5.4: DYNAMICALLY LOADED MODULES

The last major type of Assembler Language module in the Unified Graphics System is the dynamically loaded module. These modules include the character stroke generators, utility modules, error message modules, and the device-dependent modules. The example shown below is the skeleton of the device-dependent code for the IBM 2250 Display Console.

```
I502     TITLE 'UGIB502...2-ND LEVEL I/O FOR THE IBM 2250'
         PRINT ON,GEN,DATA
         SPACE 4
UGIB502  UGPRLOG MCA=YES,DDA=YES,BASE2=STM1
   ...   Main Section of Code  ...
         UGEPLOG                 RETURN TO CALLER.
         SPACE 2
         LTORG                   PROGRAM LITERALS.
         SPACE 4
         UGMCA                 MAIN COMMUNICATION AREA.
         SPACE 4
         UGDDAI50              DEVICE DEPENDENT AREA.
         SPACE 4
         UGAQE                 ATTENTION QUEUE ELEMENT.
         SPACE 4
DSAX     DSECT                 DYNAMIC STORAGE AREA.
SAVE     DS    40F               SAVE AREA.
ARGL     DS    16F               PROCESSED ARGUMENT LIST.
ARGN     EQU   ARGL                ARGUMENT LIST LENGTH.
ARGU     EQU   ARGL+4              NOT USED.
ARGX     EQU   ARGL+8              ADDRESS OF MCA.
ARGY     EQU   ARGL+12             MODULE INDEX.
ARGZ     EQU   ARGL+16             MODULE NAME.
```

```
ARG1     EQU   ARGL+24          1-ST ACTUAL ARGUMENT.
ARG2     EQU   ARGL+28          2-ND ACTUAL ARGUMENT.
ARG3     EQU   ARGL+32          3-RD ACTUAL ARGUMENT.
ARG4     EQU   ARGL+36          4-TH ACTUAL ARGUMENT.
ARG5     EQU   ARGL+40          5-TH ACTUAL ARGUMENT.
ARG6     EQU   ARGL+44          6-TH ACTUAL ARGUMENT.
ARG7     EQU   ARGL+48          7-TH ACTUAL ARGUMENT.
ARG8     EQU   ARGL+52          8-TH ACTUAL ARGUMENT.
    ...  Other Variables in DSA  ...
         DS    0D
MDSA     EQU   *-DSAX          LENGTH OF DSA.
         SPACE 4
         END
```

In this case, the prologue macro specifies that base registers must be assigned for both the MCA and the DDA. The BASE2=STM1 parameter signals that two base registers should be assigned to the program CSECT. STM1 will be a location symbol in the expansion of UGPRLOG.

This module contains DSECT's for the MCA, DDA, and AQE. The UGDDAI50 macro expands into a DSECT for the DDA with the extension for the IBM 2250.

The second item in the DSA is the argument list that is supplied to the module when it is called. The UGPRLOG moves the actual argument list into this area. Only the number of bytes specified in the argument list length are actually moved.

The structure of the character stroke generators is slightly different from the module shown above. The stroke generators have the NEWDSA=NO item in the parameter list of UGPRLOG and UGEPLOG. In addition, the argument list is not moved into the DSA, but is addressed by a DSECT.

SECTION 5.5: UTILITY MODULES

The modules described in this section are dynamically loaded at execution time and their structure is the same as the modules described in the previous section. The reason for this special section is that the utility modules, UGUTIL1 and UGUTIL2, perform a number of functions that are dependent on local modifications of the IBM 360/370 operating system.

The module named UGUTIL1 is called by a number of the device-dependent open modules. This module can determine the name of the job being executed and the output bin number. These items are added to the output on many devices for dispatching purposes. The determination of job name is a relatively system-independent operation; UGUTIL1 starts at location 16 and chains through the system control blocks until the TIOT is reached. The TIOT contains the job name. Obtaining the output bin number, however, is completely dependent on a local modification of the operating

system. The output bin number is obtained by using the WTOR macro to send messages to the ASP operating system. The replies to these inquiries eventually return the bin number.

The SUPERZAP utility program may be used to modify UGUTIL1 so that it does not try to do this SLAC-dependent operation with the cards:

```
VERIFY  0044  FFFF
REP     0044  0000
```

This modification will cause the bin number to appear as three question marks if it is requested.

The second utility module is UGUTIL2. This module is called at a number of critical points in a program. These points include (1) the time after a graphic device has been opened and before the device-dependent code is loaded into memory, (2) the time when all graphic devices have been successfully closed, and (3) the time just before a character stroke generator is loaded into memory. At the first and third points, System Management Facility (SMF) records are generated. The writing of these records is done with the SMFWTR macro [CHA-1] which allows a problem program to write records to the SMF data sets. At the first point described above, a Type 140, Sub-type 9 record is written which contains the graphic device type, open count, and programming language identification. At the third point, a Type 140, Sub-type 9 record is written which contains the identification of the character stroke generator being loaded. In addition, the header of the SMF records contains the job name and date, among other items.

The SMF records are gathered together, reformatted, and may be examined by other programs. A description of the format of these records and other useful information about them will be found in [EHR-2]. The Unified Graphics System includes a program, SMFSUMRY, which identifies all records produced by the Unified Graphics System and summarizes them.

The SUPERZAP utility program may also be used to modify UGUTIL2 so that it does not try to write these SMF records. The modification is the same as for the module UGUTIL1.

SECTION 5.6: ERROR MESSAGE MODULES

The error message modules are also loaded at execution time when needed. These modules are lists of the error messages for a single subroutine and do not contain any executable code. The name of an error message module is formed by appending the letter "X", as the seventh letter, to the name of the corresponding subroutine. The error message module for UGOPEN, named UGOPENX, is shown below.

```
OPEN    TITLE 'UGOPENX...ERROR MESSAGES FOR UGOPEN'
        PRINT ON,NOGEN,DATA
```

```
          SPACE 4
UGOPENX   UGERPRG                ERROR MESSAGE PROLOGUE.
          SPACE 2
          UGERL 1,MSG1,          LIST OF REPORTABLE ERRORS.      X
                2,MSG2,                                          X
                3,MSG3,                                          X
                11,MSG4,                                         X
                12,MSG5
          SPACE 2
MSG1      UGERM 'INVALID OR DUPLICATE  ...  THE "IDENT"', X
                'PARAMETER MUST HAVE A NO ... IF YOU ARE', X
                'TRYING TO OPEN MORE T ... DEVICE, THEN', X
                'EACH DEVICE MUST HAVE A  ... CATION.    '
          SPACE 1
MSG2      UGERM 'TOO MANY GRAPHIC DEVICES ... AN BE OPEN AT', X
                'ONCE.                    ...             '
          SPACE 1
MSG3      UGERM 'A VALID GRAPHIC DEVIC ... IFIED. CHECK', X
                'YOUR "OPTIONS" PARAMETER ...          '
          SPACE 1
MSG4      UGERM 'THE GRAPHIC DEVICE CANN ... CHECK YOUR DD', X
                'CARDS.                   ...           '
          SPACE 1
MSG5      UGERM 'NOT ENOUGH DISPLAY BUF ... AILABLE. YOU', X
                'WILL PROBABLY HAVE TO BE ... FIL" OPTIONS', X
                'ITEM.                    ...           '
          SPACE 4
          END
```

The lines of code defining the actual error messages have been modified so they fit on a single line of this document.

It is more convenient to start at the end of this module and work one's way to the beginning in explaining it. The UGERM macro is used to define the error message itself. The error message may consist of an arbitrary number of lines of 52 characters each. The macro itself counts the number of lines in the error message. The UGERL macro is used to define the list of valid errors. Each error is defined by giving its index and a symbol where the error message is found. The expansion of this macro includes a count of the number of lines in each error message. This macro also counts the number of error messages. The final macro in use here is the UGERPRG macro. The expansion of this macro contains the number of error messages. The result of these macros is a series of tables which can easily be scanned to locate any required error message.

The subroutine which prints the error message is UGX003. When this subroutine is ready to print the information in the error message modules, it calls subroutine UGX004 to obtain the error message a line at a time. Each time UGX004 returns a line of text, subroutine UGX003 prints the line and calls UGX004 again.

The first time subroutine UGX004 is called, it checks to see if an error message module is available. If a module with the correct name is available, it is loaded into memory. UGX004 then searches for the required error message and, if found, returns

the first line to the calling subroutine. On subsequent calls, UGX004 returns the other lines in the error message. When UGX004 is called and there are no more lines available in the error message, or if the error message was not defined, then UGX004 deletes the error message module from memory.

Thus, at most one error message module is ever in memory at one time. Also, if an error message module is not available, or if the error message is not defined, then nothing disastrous happens. In this case, the printed error message contains the subroutine name and error index, but no explanation of the error is printed.

SECTION 5.7: RECURSIVE ERROR PROCESSORS IN FORTRAN

This section does not describe any module within the Unified Graphics System, instead it describes how a recursive error processing subroutine for FORTRAN may be written. An error processing subroutine may call subroutines in the Unified Graphics System. If one of these subroutines detects an error, the error processing subroutine will have to be entered recursively. Normally, the FORTRAN version of the Unified Graphics System does not let this happen but, instead, issues a terminal error message. Since FORTRAN programs on the IBM 360/370 computers are not recursive, a recursive error processing subroutine must be written in Assembler Language. The skeleton of such a subroutine is shown below.

```
XERR     TITLE 'UGXERR...ERROR PROCESSING SUBROUTINE'
         PRINT ON,GEN,DATA
         SPACE 4
UGXERR   UGPRLOG MCA=YES       INITIALIZE PROGRAM.
         MVC   ARGL(12),0(1)   OBTAIN THE ARGUMENT LIST.
         MVI   MCAFTNER,MCAFTNE1  ENABLE RECURSIVE CALLS.
    ...  Main Section of Code  ...
*  INDICATE THAT THE ERROR HAS BEEN PROCESSED.
DONE     L     A0,LEVL          GET ADDRESS OF LEVEL.
         XC    0(4,A0),0(A0)    SET LEVEL TO ZERO.
         SPACE 2
*  TERMINATE PROCESSING AND RETURN TO CALLING PROGRAM.
RETN     UGEPLOG                RETURN TO CALLER.
         SPACE 2
         LTORG                  PROGRAM LITERALS.
         SPACE 4
         UGMCA                MAIN COMMUNICATION AREA.
MCAFTNER EQU   MCALAGDP          RECURSION PERMITTED FLAG.
MCAFTNE1 EQU   X'FF'             RECURSION PERMITTED VALUE.
         SPACE 4
DSAX     DSECT                DYNAMIC STORAGE AREA.
SAVE     DS    40F              SAVE AREA.
ARGL     DS    3F               INPUT ARGUMENT LIST
LEVL     EQU   ARGL               ADDRESS OF LEVEL.
NAME     EQU   ARGL+4             ADDRESS OF NAME.
```

```
INDX     EQU   ARGL+8              ADDRESS OF INDEX.
    ...  Other Variables in DSA  ...
         DS    0D
NDSA     EQU   *-DSAX          LENGTH OF DSA.
         SPACE 4
         END
```

The subroutine, as shown, uses the macros UGPRLOG and UGEPLOG as the entry and exit process. These macros allow the subroutine to be recursive. The first thing the subroutine does is to save its arguments and set HCAFTHER to the value HCAFTHE1. Actually, HCAFTHER only needs to be set to this value once, but it is usually easier to do it each time as shown. Following this statement, the subroutine should check both LEVL and NAME for each error it is ready to handle. If error recovery is complete, the level of the error should be set to zero with code similar to the instructions shown above. The macro libraries WYL.CG.RCB.UGCOMMAC and WYL.CG.RCB.UGFTHMAC must both be made available when a subroutine like that described here is assembled.

This section describes some of the problems involved in extending the system to other graphic devices, programming languages, or host computers.

SECTION 6.1: ADDING A NEW GRAPHIC DEVICE TO THE SYSTEM

The steps required to add a new graphic device to the Unified Graphics System should be reasonably clear from the preceding sections. This section will review these steps and will provide some additional information.

The procedure for adding a new graphic device to the system is:

1. The module UGOPEN should be changed. The source for this module is in WYL.CG.RCB.UGSOURCE and the change consists of adding a new entry to two tables. The first table is a list of the valid device types. The second table is a list of the names of the device-dependent open modules. When the source for UGOPEN has been modified, it must be re-compiled for each of the supported languages. These trivial changes are the only modifications which must be made to existing code.
2. A macro which defines the extension to the DDA for the new device must be prepared. This macro should be put into the library WYL.CG.RCB.UGRUNMAC.
3. The device-dependent modules must be written. The source for these modules should be added to WYL.CG.RCB.UGSOURCE and the compiled modules go into WYL.CG.RCB.UGRUNLIB. For most devices, the device-dependent modules will consist of exactly three modules. The first module does the device-dependent open operations, the second is the device-dependent code which remains in memory from the time that the device is opened until it is closed, and the third module does the device-dependent close operations.

The Unified Graphics System already supports a wide range of graphic devices. The problem of adding a new device can usually be solved by selecting the DDA extension macro and device-dependent code from an already supported device of similar capability, and modifying this code to support the new device.

Although it would initially appear that the first step in the preceding list must be done before it is possible to execute new device-dependent code, this is not the case. A special item, which is not documented in the Unified Graphics Programming Manual [BBA-1], is available in the options list of UGOPEN, which allows experimental device-dependent code to be executed without changing UGOPEN. This item is XSYSTEM=<value> where <value> is

the name of the device-dependent open module. Thus, the item XSYSTEM=UGIB501 has exactly the same effect as the item IBM2250 in the options list of UGOPEN.

The size of the device-dependent modules depends on the complexity of the graphic device. Typically, the first and third modules are quite small; the first is usually 500 to 1500 bytes in size, while the third is less than 500 bytes. The length of the second module is larger. For non-interactive devices, it ranges from 2000 to 3500 bytes, and for an interactive device, it is usually between 4500 and 8000 bytes.

SECTION 6.2: ADDING A NEW PROGRAMMING LANGUAGE TO THE SYSTEM

The difficulty in adding a new programming language to the Unified Graphics System depends strongly on the specific programming language in question. A language similar to FORTRAN or PL/1 would probably be reasonably easy; a language distinctly different from these could pose substantial problems.

While most of the modules in this system are written in Assembler Language, a number of them are written in the high level language. The modules written in the high level languages are: UGAXIS, UGXHCH, UG3DMS, UGCNTR, UGRSCL, UGDXDY, UGPROJ, UGORTH, UG3TO2, UGCNVF, and UGX003.

The modules written in Assembler Language should be easy to convert to a new language. The basic source modules themselves should not require any changes whatsoever; only the macros used by these modules need to be changed. As examples of the size of this problem, the FORTRAN macro library contains about 850 cards while the PL/1 macro libraries contain about 1000 cards. Actually, more than half of these cards are comment and macro skeleton cards; only 400-500 cards constitute actual machine instructions, and it is principally these cards which must be examined and possibly modified. In addition, there are two Assembler Language modules which are very language-dependent and will have to be re-written; the modules are UGX002 and UGX201. In summary, the conversion problem for Assembler Language modules is much simpler than the conversion problem for modules written in the high level languages.

The order that the modules are converted can significantly affect the amount of work done. The proper sequence will allow the modules to be checked-out individually before they must work together as a system. A sequence that has proven effective is:

1. The language-dependent extension of the HCA must be defined and the module UGX000 must be compiled.
2. The options scanning modules (UGOPTN and UGX001) should be converted and checked-out. This step will require that the UGPRLOG and UGERLOG macro be prepared, and may require that UGX201 also be written.
3. The error processing modules (UGRERR, UGXERR, UGX002,

UGX003, and UGX004) can then be converted. Other modules like UGX301 may have to be converted at this time.

4. Next, the element generation modules (UGEINT, UGEPNT, UGELIN, etc., and UGX121) may be converted and checked-out. Converting UGX121 will require that the UGPRLOG and UGEPLOG macros with the NEWDSA=NO option be available.
5. The remainder of the internal modules (UGX101, etc.) may now be compiled. With the exception of UGX113, which requires two new macros, nothing new is required in this step. It is not really necessary to check out these modules at this time; there are very few things which can go wrong at this step.
6. The basic set of I/O modules (UGOPEN, UGCLOS, UGEPUT, and UGPICT) may be converted and checked-out by generating pictures for a non-interactive device. If UGX201 has not been converted earlier, it will now be necessary to convert it.
7. The additional I/O modules (UGCTRL, UGEATN, UGDATN, UGRATN, UGKPUT, and UGKGET) should be converted and checked-out on an interactive device.
8. The final step is to convert the remaining modules. The order of doing this is not significant since the remaining modules are relatively independent of each other.

The first language supported by the Unified Graphics System was the PL/1 Optimizing Compiler. Next, the FORTRAN version was created and, later, a PL/1 F-Level Compiler version was generated. The order described above was used in preparing the FORTRAN and PL/1 F-Level versions. The fact that these conversions were reasonably trouble-free reinforces the belief that the Assembler Language modules are coded in a language-independent manner.

SECTION 6.3: TRANSFERRING THE SYSTEM TO ANOTHER HOST COMPUTER

The job of transferring the Unified Graphics System to another host computer is a substantial problem. To support the full Unified Graphics System requires a reasonably sophisticated operating system. The Unified Graphics System makes significant use of the GETMAIN, FREEMAIN, LINK, LOAD, DELETE, and other macros on the IBM computers. The operating system on any computer supporting the full Unified Graphics System would have to supply the equivalent of these functions. Also, the ability to write recursive modules is necessary. The processing of errors by a user's program will often cause the device-dependent code to be used recursively. The handling of attentions on an interactive device can also cause recursive use of the device-dependent code.

Restricted versions of the Unified Graphics System can be implemented on computers which do not have all the facilities required for the full Unified Graphics System. For example, if the LINK, LOAD, and DELETE macros are not available, it is still possible to have a Unified Graphics System which does not support dynamically selected or multiply open graphic devices; the graphic device would have to be selected at LINK-EDIT time. Multiple character stroke generators will also not be dynamically selectable. Without the GETMAIN and FREEMAIN macros, it will probably be necessary to restrict the error processor. It may also be impossible to queue attentions from interactive devices; UGRATN would always act as if the LAST options item was given.

A version of the Unified Graphics System has been prepared which is coded almost completely in FORTRAN. This version of the system is described in [BEA-6] and has all of the restrictions described in the previous paragraph. C. Granieri of the Stanford Computation Center, and some people working with him, have modified this version so that it runs on the PDP-11/55 computer associated with the Crystal Ball experiment at SLAC.

The algorithms in most of the modules in the Unified Graphics System are quite transparent. However, a few algorithms require additional explanation. Those explanations are in the following sections.

SECTION 7.1: SUBROUTINE UGXHCH

The algorithm used in subroutine UGXHCH is basically the following:

1. A family of parallel lines are set up which define the cross-hatching.
2. The lines in this family which actually intersect the region to be cross-hatched are determined.
3. For each of these lines, the following operations are performed:
   A. The line of cross-hatching is intersected with the region boundary to obtain all of the intersection points.
   B. The intersection points are sorted so that they are in order along the line. This order alternates, in direction along the line, in an attempt to optimize the data for drum type plotters.
   C. At this point, there will be an even number of points available and they are output by blanking to the first point, drawing to the second point, blanking to the third point, etc.

The data which describes the lines of cross-hatching are:

- XCRD The X coordinate of a point through which a line of cross-hatching will pass.
- YCRD The Y coordinate of a point through which a line of cross-hatching will pass.
- ANGL The angle at which the lines of cross-hatching will be drawn.
- SPAC The spacing between the lines of cross-hatching.

This data is shown in Figure 7.1.1.

First, the equations of the lines of cross-hatching will be developed. A unit vector along a line of cross-hatching is:

[cos(ANGL),sin(ANGL)]

Therefore, a unit vector perpendicular to one of these lines is:

[-sin(ANGL),cos(ANGL)] (1)

A reference point on each line of cross-hatching may be obtained by moving along Vector (1) in integral multiples of SPAC from the point (XCRD,YCRD). Therefore, a point on each line of cross-hatching is given by:

[XCRD-I*SPAC*sin(ANGL),YCRD+I*SPAC*cos(ANGL)]

Where I takes on all integral values. Thus, the parametric equations of the family of lines forming the cross-hatching are:

```
X = cos(ANGL)*t + [XCRD-I*SPAC*sin(ANGL)]
Y = sin(ANGL)*t + [YCRD+I*SPAC*cos(ANGL)]
   where I=...,-1,0,1,2,...
```

These equations may also be displayed as:

```
X = AX*t + BX+I*CX = AX*t + DX
Y = AY*t + BY+I*CY = AY*t + DY                                    (2)
   where I=...,-1,0,1,2,...
```

and the A's, B's, and C's are constants and the D's depend only on the line of cross-hatching under consideration.

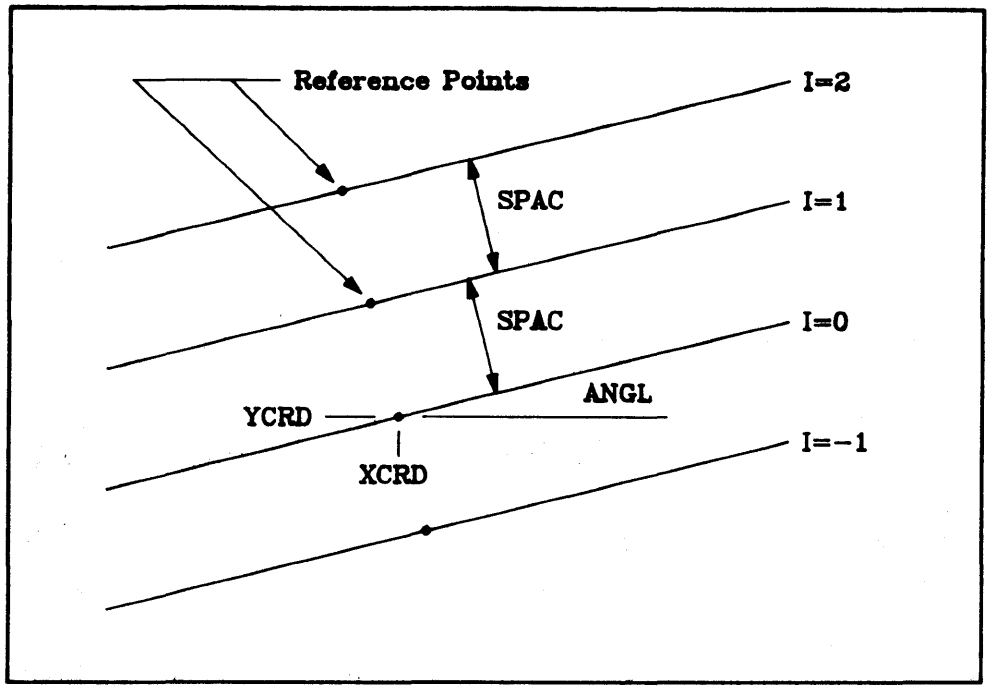

Figure 7.1.1: The Lines of Cross-Hatching.

The next problem is to determine which of the lines in Equations (2) actually intersect the region. To do this, we compute the (non-integral) value of I in Equations (2) which determines a line passing through each point in the definition of the region boundary. By computing the minimum and maximum values of this set of numbers and rounding up or down to an integer, we obtain the values of I, which bound the region to be cross-hatched. The computation for I is done by eliminating t in Equations (2), solving for I, and simplifying. The result is:

```
I = [AX*(Y-BY)-AY*(X-BX)]/SPAC
```

The last problem to be considered is that of computing the intersections of the line of cross-hatching with the region boundary. To do this, consider the function:

```
D(X,Y) = AY*X - AX*Y + (AX*DY-AY*DX)
```

The value of this function is zero if the point (X,Y) is on the line, is positive on one side of the line, and is negative on the other side. Further, abs(D(X,Y)) is the distance from the point (X,Y) to the line of cross-hatching. By testing each adjacent

pair of points on the boundary, in turn, to find a place where the function D gives a change in sign, we find the position of all possible intersections between the region boundary and the line of cross-hatching. Suppose a pair of points are (X1,Y1) and (X2,Y2) and the respective values of D(X,Y) are D1 and D2. Then the point of intersection is given by:

```
XI = (abs(D1)*X2+abs(D2)*X1)/(abs(D1)+abs(D2))
YI = (abs(D1)*Y2+abs(D2)*Y1)/(abs(D1)+abs(D2))
```

The point determined by these equations is, except for round-off error, on the line of cross-hatching. The next step is to determine the value of t in Equations (2) corresponding to (XI,YI). The value of t which gives the closest point on the line of cross-hatching to the point (XI,YI) is given by:

```
t = AX*(XI-DX) + AY*(YI-DY)
```

After all such t's are computed, they are sorted into ascending or descending order, and the points are re-computed from the value of t using Equations (2), and added to the graphic element by calling subroutine UGELIN.

SECTION 7.2: SUBROUTINE UG3DHS

The algorithm used in subroutine UG3DHS is basically very simple. Its implementation, however, does get somewhat complicated. To begin, consider the case where the lines on the surface are to be drawn in only one direction and assume that this direction is roughly perpendicular to the viewing direction. Also, suppose that a picture of the upper side of the surface is to be developed. The first step is to project the surface line which is closest to the observer into the drawing plane. This line may be drawn in full because there is nothing closer to the observer which could hide it. This line now becomes a "height function", as shown in the picture on the left in Figure 7.2.1. This height function has a value of minus infinity outside the domain of the projected line. The next step in developing the picture is to project the next surface line into the picture plane. Only those parts of the second line that are above the height function are actually drawn. The height function is then modified so that it represents the highest point of either line. In the picture on the right in Figure 7.2.1, the actual height function is shown by the solid lines. Notice that this height function can be a very jagged function. Successive lines in the surface may be drawn in the same manner.

In determining what part of a projected surface line is above the height function, two simplifying assumptions are made:

1. If both end points of one of the short straight line segments is below the height function, then the entire segment is assumed to be below the height function and the line segment is not drawn.
2. If both end points are above the height function, the entire segment is assumed to be visible and it is drawn.

When exactly one end point is above the height function, then the actual intersection with the height function is determined and a line is drawn from the visible point to the point of intersection.

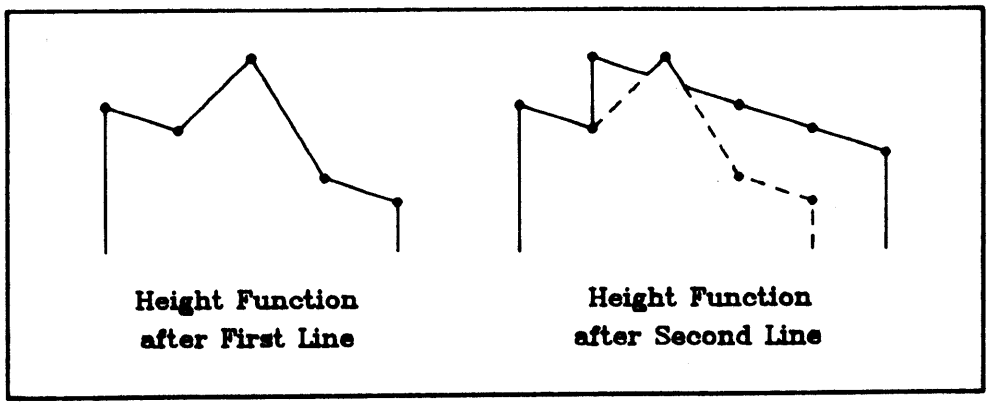

Figure 7.2.1: Development of the Height Function.

It is a common misconception that a complete picture of a mesh surface is produced by first processing the lines in one direction, and then, independently, processing the lines in the other direction. Figure 7.2.2 demonstrates that this assumption is false.

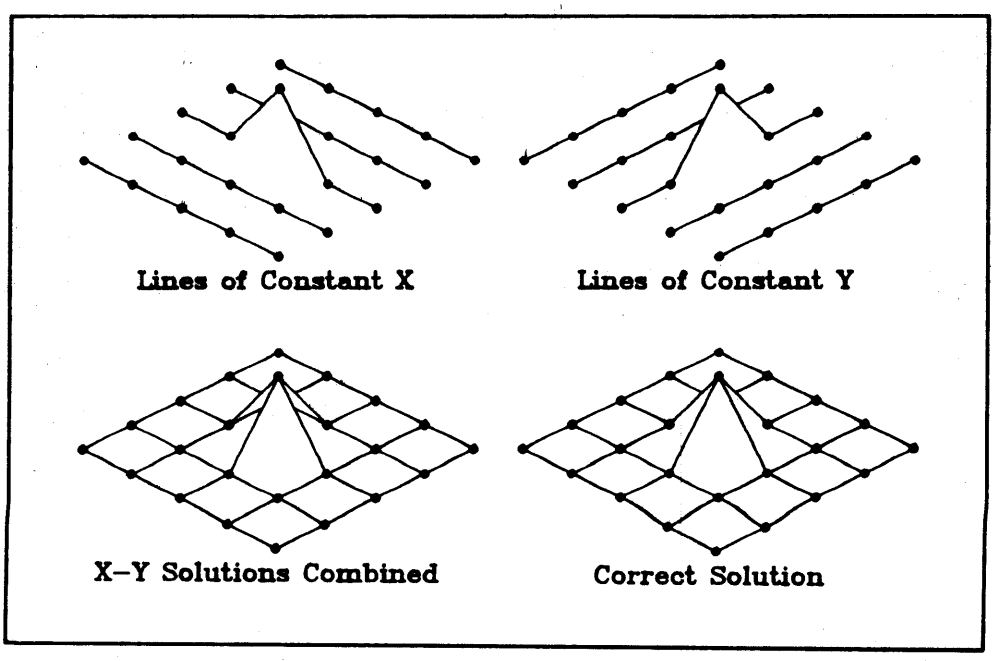

Figure 7.2.2: Combining the X-constant and Y-constant Lines.

In view of the above discussion, the complete algorithm may be stated as follows:

1. The mesh surface is examined to determine which edge is closer to the viewer. Suppose these lines are the

V2=constant lines where V2 is either X or Y. The lines in the other direction are the V1=constant lines.

2. The height function is initialized to a constant with a value of minus infinity.
3. For each V2=constant line, starting at the line closest to the viewer, the following is done:
   A. Each straight line segment on the V2=constant line is examined. The part of it that is above the height function is drawn and the height function is modified to include this new information.
   B. If this is not the last V2=constant line, then the segments of the V1=constant lines between the current V2=constant line and the next V2=constant line are examined. The part of these line segments that is above the height function is drawn and the height function is modified.

When the bottom side of the mesh surface is being developed, the height function is initialized to plus infinity, and lines are drawn only when they are below the current height function.

The height function itself is contained in the array WKAREA in the calling sequence to subroutine UG3DHS. The function is saved as a list of (X,Y) coordinates of the break points in the height function. As Figure 7.2.1 shows, some of the segments in the height function are vertical. To contain the height function, the array WKAREA is divided up into three word segments. Each segment contains two halfword fixed point values and two floating point values, as follows:

- HFFP A halfword fixed point value which gives the index in WKAREA of the start of the three word segment containing the (X,Y) coordinate of the next point of the height function in a leftward direction.
- HFBP A halfword which gives the index of the start of the three word segment containing the (X,Y) coordinate of the next point in a rightward direction.
- HFXC The X coordinate of the point on the height function.
- HFYC The Y coordinate of the point.

The rightmost point of the height function is always in WKAREA(1)...WKAREA(3) and the leftmost point is in WKAREA(4)...WKAREA(6). The height function is thus maintained as a concatenation of points defining the break points. By maintaining the function as a doubly linked list, the many modifications that must be made to this function can be accomplished in a reasonably fast time.

The algorithm used in subroutine UG3DHS is based on the information in [WRI-1, BAR-1]. Other algorithms of this nature have been described in [KUB-1, WIL-1, WAT-1].

SECTION 7.3: SUBROUTINE UGCNTR

An extremely simple algorithm can be written to produce contour plots. For each contour line, each rectangular surface patch is

examined. For each surface patch, the four boundaries are examined. If one corner point of a boundary line is above the contour value and the other is below the contour value, then the intersection of the contour line and the boundary is determined by linear interpolation. As long as exactly two intersection points per surface patch are found, they can be joined together. The collection of all of these lines will form a contour plot of the surface. Unfortunately, there are two things wrong with this simple algorithm. First, most graphic devices work more efficiently when the contour lines are drawn as a sequence of concatenated line segments instead of drawing them in some arbitrary order (think of all the extra "pen-up", "pen-down", and movement with the pen up on a drum plotter). The second problem is associated with the Unified Graphics System itself; line structure cannot successfully be applied to a curve that is not drawn as concatenated line segments. The reason is that line structure must start over after each blank vector, and the dashes (for example) will not be evenly spaced along the entire length of the line. These reasons show that it is important to generate the contour lines as a series of concatenated line segments.

Within subroutine UGCNTR, there are a few basic variables which must be understood. The first of these are IROW and ICOL. These variables specify the "row" and "column" of the surface patch under consideration. IROW takes on values from 3 to NDIM, and ICOL takes on values from 3 to NDIM. Thus, the indices of the Z values of the corner points of the surface patch are as shown in Figure 7.3.1. This figure also gives the values of ISID, the surface patch side index.

The array WKAREA is used to contain a flag for each surface patch boundary in the surface. These flags can take on the value "processed" or "unprocessed". The details of how these flags are stored will be given later. For now, the reader should keep in mind that the ISID=2 side of the (IROW,ICOL)-th surface patch is the same line segment as the ISID=0 side of the (IROW,ICOL+1)-th surface patch. Similarly the ISID=3 side of the (IROW,ICOL)-th surface patch is the same as the ISID=1 side of (IROW+1,ICOL)-th patch.

The basic algorithm can then be stated as follows:

1. A contour line is selected and the flags in WKAREA are set to unprocessed.
2. The line segments along the boundary of the whole surface are checked to see if they are marked unprocessed and if the contour line intersects the line segment. If the answer is no, the line segment is marked processed. If the answer is yes, the curve is followed until it terminates at another boundary line segment and all line segments that are encountered are marked processed.
3. The ISID=0 line segment of each of the surface patches is checked to see if it is marked unprocessed and if the contour line intersects the line segment. If the answer is no, the line segment is marked processed. If the answer is yes, the curve is followed until it terminates at the initial line segment and all

encountered line segments are marked processed.
Step 2 generates all contour lines which begin and end on the surface boundary while Step 3 generates all closed curves. By checking the boundary first, we can be certain that these curves will be drawn as a single concatenated curve. Notice, in Step 3, that it is <u>not</u> necessary to check all four boundaries of each surface; the ISID=0 side is enough since any closed curve must cross a ISID=0 side of some surface patch.

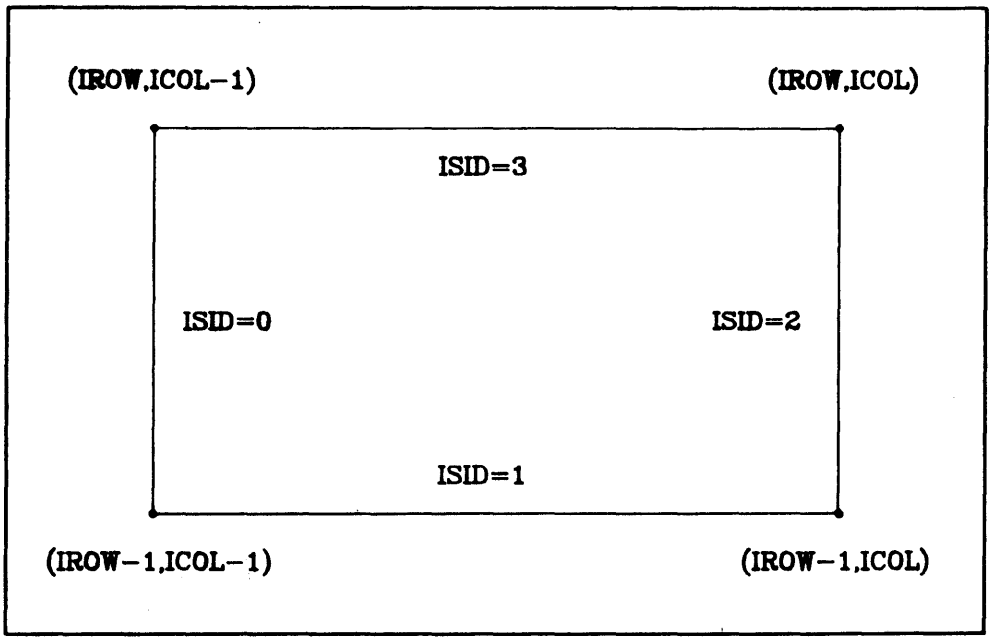

Figure 7.3.1: The Indices IROW, ICOL, and ISID.

The problem of following a contour line is reasonably straightforward. The only problem comes when the number of sides intersected by the contour line is four. It is easy to see that the number of intersections is always zero, two, or four. The results for zero or two intersections are obvious, but four intersections present three possibilities, as shown in Figure 7.3.2. When a surface is defined by a mesh of points, as in this subroutine, there is not enough information available to decide between these possibilities. In this algorithm, we choose the third configuration because it is the most symmetric. This choice is very simply accomplished in the algorithm; if a surface patch is entered by the ISID-th side, the side opposite this side is checked for the exit side first before the adjacent sides are checked. The side opposite the ISID-th side is the MOD(ISID+2,4)-th side. The adjacent sides are the MOD(ISID+1,4)-th and MOD(ISID+3,4)-th. Thus, following the curve is a simple process.

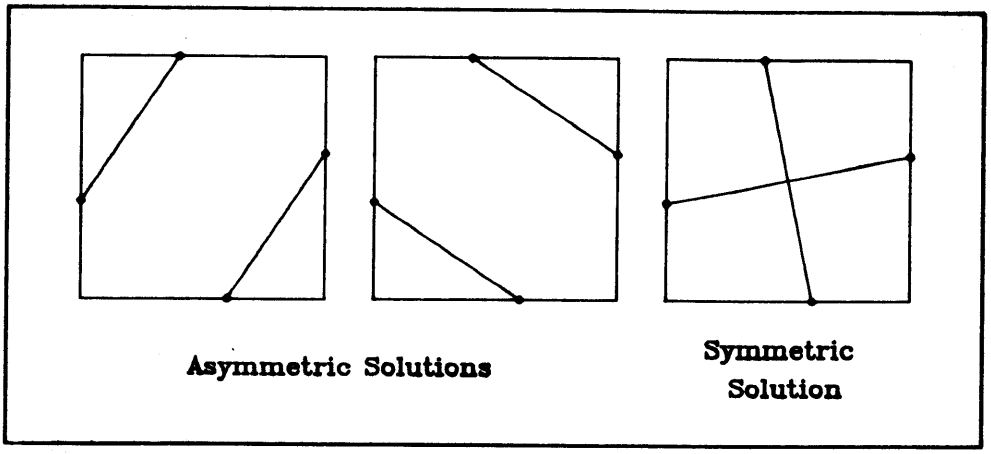

Figure 7.3.2: Ambiguities with Four Intersections.

The only thing still to be discussed is the manner that the "processed" and "unprocessed" flags are stored in WKAREA. The flags are stored as a single bit, 0 meaning unprocessed and 1 meaning processed, in WKAREA with 30 bits per word. The bits are packed and unpacked by multiplying and dividing by powers of two. To explain how an individual bit is found, suppose we wish to find the bit for the (IROW,ICOL,ISID)-th line segment. If ISID is two, we reset ISID to zero and increase ICOL by one. If ISID is three, we reset ISID to one and increase IROW by one. The bit number, starting with zero, of the bit that must be accessed is then:

```
  NBIT = 2*[ (IROW-3)*(NDIM-1)+(ICOL-3) ] + ISID                (1)
```

Then let:

```
  NWRD = 1 + NBIT/30
  MBIT = 1 + MOD(NBIT,30)
```

Then the required bit is the MBIT-th bit of WKAREA(NWRD). Notice that Equation (1) shows that the actual number of words required in WKAREA is:

```
  [ (NDIM-2)*(NDIM-1)+(NDIM-2)+15 ]/15
```

The number MDIM*NDIM/15 that is given in the Unified Graphics Programming Manual [BEA-1] is only an approximation that is correct for all cases except when MDIM or NDIM is quite small. By using this method of counting the bits, there are a few bits that do not actually get used. The PL/1 version could be re-written to use a bit string for WKAREA; such a procedure would probably be more efficient than the current version. This was not done because the improvement would be slight and it was thought to be more important to retain the similarity of the FORTRAN and PL/1 versions.

An algorithm very similar to the one used here is described in [COT-1]. I was also able to examine a listing of CERN's contour plotting subroutine in GD3 [MIL-1] and the idea of searching the boundary first and then the interior came from there.

SECTION 7.4: SUBROUTINE UGDXDY

The problem addressed by subroutine UGDXDY is to take a given range on an axis and expand that range a minimal amount and divide it into a number of equal length segments, such that each segment begins and ends on a "round number". For the purposes of this subroutine, a "round number" is defined as a number of the form:

```
N*(P*10**K)
```

where N, P, and K are integers and P is one of the set {1, 2, 5}. The values of P and K are fixed for all labels and the factor in parenthesis is the segment length. The actual labels correspond to a series of consecutive values for N.

A simple algorithm for determining the labels according to these constraints when the number of segments is given, is described in [DIX-1]. If S is the number of segments required, then a first approximation for K is:

```
K = [LOG10((HIDATA-LODATA)/S)]
```

where [X] is the greatest integer in X. If ((HIDATA-LODATA)/S)/10**K is greater than the largest of the P's, then K is incremented by 1. Then the smallest P is found which is greater than or equal to ((HIDATA-LODATA)/S)/10**K. This procedure gives a starting value for P and K. Now let:

```
s = P*10**K
LOLAB = [((HIDATA+LODATA)/2-S*s/2)/s]*s
HILAB = LOLAB+s*S
```

Thus LOLAB and HILAB are round numbers, the distance between them is divided into S segments of length s, and s is also a round number. However, there is no guarantee that LOLAB and HILAB bracket the data. The final step is, therefore, to check that the data is bracketed. If it is not, two things may be tried: first LOLAB and HILAB may be incremented by s, or if this does not work, P and K must be increased to the next permissible values.

The basic addition to the algorithm described in [DIX-1] is that the algorithm is applied to a range of segment counts. The algorithm is applied for all segment counts between (MINLAB-1) and (MAXLAB-1). Subroutine UGDXDY then selects the segment count which minimizes the expansion of the range.

Other algorithms of this nature have been described in [GIA-1, LEW-1].

SECTION 7.5: SUBROUTINES UGPROJ, UGORTH, AND UG3TO2

The basic product of both subroutine UGPROJ and UGORTH is a 3x4 matrix which defines a projective transformation from 3-space into 2-space. The orthogonal transformation of subroutine UGORTH is just a special case of a projective transformation where the projection point has been moved to infinity. The 3x4 matrix, M, transforms a point P in 3-space with the coordinates (X,Y,Z) into

a point Q in 2-space with the coordinates (t,u). The coordinate transformation is carried out by:

$$L*\begin{bmatrix} t \\ u \\ 1 \end{bmatrix} = M*\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \qquad (1)$$

In subroutine UG3TO2, Equation (1) is expanded into three scalar equations for L*t, L*u, and L. The third value is divided into the first two to obtain t and u.

First, we shall derive the form of the matrix, M, for a projective transformation as generated by subroutine UGPROJ. Figure 7.5.1 shows the data used in this derivation. In the following, we shall assume that VDIR, HDIR, and UDIR are mutually perpendicular unit vectors. From Figure 7.5.1, it is evident that:

```
  E = REFP + EYED*HDIR                                          (2)
  Q = REFP + SCRD*VDIR + t'*HDIR + u'*UDIR
```

Where t' and u' are related to t and u by:

```
  t' = TVR1*(t-XBAR)                                            (3)
  u' = TVR2*(u-YBAR)
```

and TVR1, TVR2, XBAR, and YBAR are given by:

```
  TVR1 = SCRZ/(XHI-XLO)
  TVR2 = SCRZ/(YHI-YLO)                                         (4)
  XBAR = (XHI+XLO)/2
  YBAR = (YHI+YLO)/2
```

Since E, P, and Q lie on a straight line, the vectors (P-E) and (Q-E) are parallel and, therefore, are proportional. Let the constant of proportionality be L, so that we have:

```
  L*(Q-E) = (P-E)                                               (5)
```

Substituting Equation (2) into Equation (5) gives:

```
  L*[(t'-EYED)*HDIR + u'*UDIR + SCRD*VDIR] = (P-E)
```

However, this equation may be written in matrix form as:

$$L*M1*\begin{bmatrix} t'-EYED \\ u' \\ SCRD \end{bmatrix} = (P-E) \qquad (6)$$

where the columns of M1 are HDIR, UDIR, and VDIR, and P and E are considered to be column vectors. Since the 3x3 matrix M1 is orthogonal, its inverse is equal to its transpose. Thus, multiplication of Equation (6) by the inverse of M1, and using Equation (3) gives:

$$L*\begin{bmatrix} TVR1*(t-XBAR)-EYED \\ TVR2*(u-YBAR) \\ SCRD \end{bmatrix} = M2*(P-E) \qquad (7)$$

where the rows of M2 are HDIR, UDIR, and VDIR. The column vector on the left-hand side of Equation (7) may be factored into:

$$\begin{bmatrix} TVR1 & 0 & -(TVR1*XBAR+EYED) \\ 0 & TVR2 & -TVR2*YBAR \\ 0 & 0 & SCRD \end{bmatrix} * \begin{bmatrix} t \\ u \\ 1 \end{bmatrix} \qquad (8)$$

Next, the inverse of the 3x3 matrix in (8) may be written as:

$$\begin{bmatrix} 1/TVR1 & 0 & TVR3 \\ 0 & 1/TVR2 & TVR4 \\ 0 & 0 & 1/SCRD \end{bmatrix} \qquad (9)$$

where TVR3 and TVR4 are defined by:

TVR3 = (TVR1*XBAR+EYED)/(TVR1*SCRD)

TVR4 = YBAR/SCRD

Now Equation (7) may be multiplied through by Matrix (9) to obtain:

$$L * \begin{bmatrix} t \\ u \\ 1 \end{bmatrix} = M3*(P-E) \qquad (10)$$

where M3 is Matrix (9) times M2. Finally, Equation (10) may be put in the form of Equation (1) by writing:

M = (M3|(-M3*E))

This last equation states that the first three columns of M are the same as those of M3, while the last column of M is the column vector -M3*E.

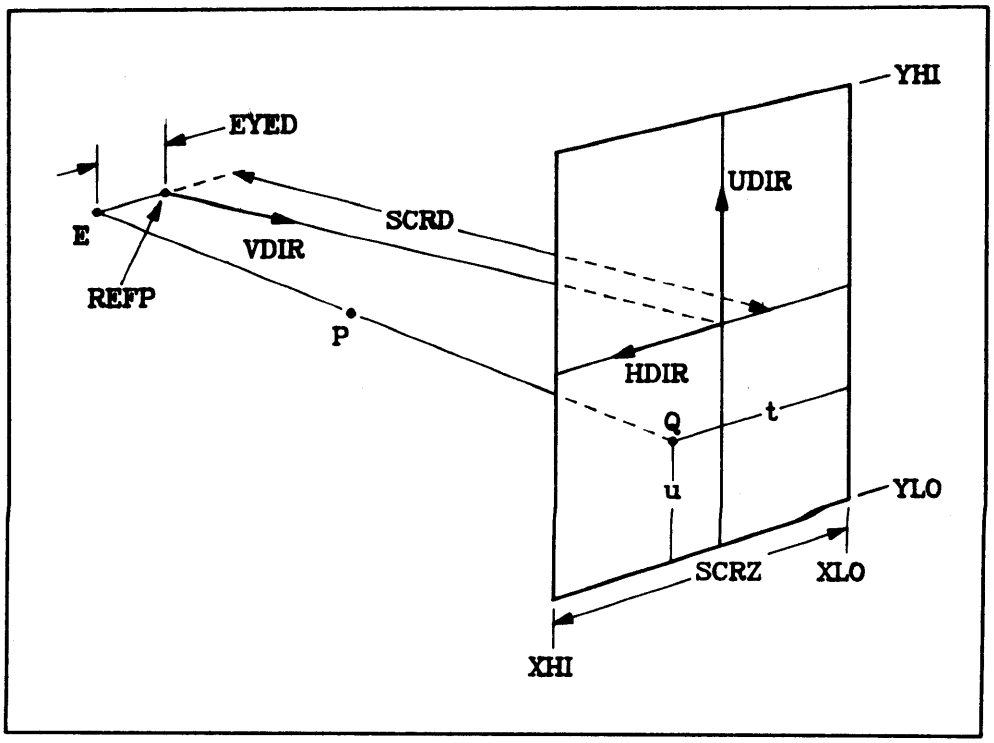

Figure 7.5.1: The Definition of a Projective Transformation.

The derivation for the transformation produced by UGORTH is somewhat simpler. In an orthogonal projection, the projection is through P, parallel to VDIR. Thus, the vector (P-Q) is parallel to VDIR and, therefore, proportional to it. Let the constant of proportionality be K so that we have:

$$P-Q = K*VDIR \quad (11)$$

$$Q = REFP + SCRD*VDIR + t'*HDIR + u'*UDIR$$

and Equations (3) and (4) remain valid. From Equations (11), (3) and (4), we get:

$$P - REFP = [K+SCRD]*VDIR + [TVR1*(t-XBAR)]*HDIR + [TVR2*(u-YBAR)]*UDIR \quad (12)$$

Taking the dot product (•) of Equation (12) with HDIR and then with UDIR, and solving the resulting equations for t and u, gives:

$$t = P•HDIR/TVR1 + [XBAR-REFP•HDIR/TVR1] \quad (13)$$

$$u = P•UDIR/TVR2 + [YBAR-REFP•UDIR/TVR2]$$

If the third row of the matrix M is set to (0, 0, 0, 1), which forces L=1, then Equation (13) may easily be put into the form of Equation (1).

The array TRANS, which is generated by UGPROJ and UGORTH, contains other information in addition to the matrix M. The complete definition of the content of TRANS is as follows:

- TRANS(1)...TRANS(12) The transformation matrix stored by rows.
- TRANS(13)...TRANS(15) The coordinates of REFP.
- TRANS(16)...TRANS(18) A unit vector in the direction of HDIR. This vector has been adjusted so that it is perpendicular to VDIR.
- TRANS(19)...TRANS(21) A unit vector in the direction of VDIR.
- TRANS(22)...TRANS(24) A unit vector perpendicular to VDIR and HDIR which defines the upward direction of the screen.
- TRANS(25) The value of EYED.
- TRANS(26) The value of SCRD.
- TRANS(27) The value of SCRZ.
- TRANS(28) The value of XLO.
- TRANS(29) The value of XHI.
- TRANS(30) The value of YLO.
- TRANS(31) The value of YHI.

Subroutine UGPROJ only makes use of the transformation matrix, but subroutine UG3DHS makes use of REFP and VDIR to decide how the surface is to be generated.

SECTION 7.6: THE LINE SCISSORING MODULE

The line scissoring module, UGX112, is one of the most basic internal modules in the Unified Graphics System. It will normally be called during the process of converting a line segment to device-dependent orders. The line segment may be those that have been added to a graphic element by subroutines UGELIN or UGELNS, or the line segments may be those that are

generated by the character stroke generators.

This scissoring module operates on end points of line segments that are expressed in the device coordinate system. It is used as follows:

1. The scissoring module is called with an indication that it is to initialize itself.
2. The scissoring module is called with the coordinates of an end point of a line segment and its blanking bit.
3. The scissoring module is called with an indication that a scissored point and its blanking bit is needed.
4. If Step 3 returns a point, then it is repeated to try to obtain another point. If Step 3 did not return a point, then Step 2 is executed.

Notice that after a program passes a single point to the scissoring module, it must be ready to loop in Step 3 to obtain a number of points. The number of points which may be returned by repeated executions of Step 3 may be zero, one, or two. Zero points may be returned, for example, if the line segments being supplied in Step 2 are all outside the scissoring window. A single point will be returned in Step 3 if all of the points are inside the scissoring window. An example of a case where two points will be returned is the following: suppose two points are given which both lie outside the scissoring window, but the line segment between the points intersects the window. Then Step 3 will return two points after the second point has been supplied in Step 2.

The principal consideration in the design of this module was that it must be as efficient as possible in the case where no scissoring is required. The module always saves the last two points supplied to it. It takes only four simple comparisons to determine that a point is inside or outside the scissoring window. If the last two points supplied to the scissoring module are both inside the scissoring window, then the latest point can be returned unaltered when a point is requested.

It is only when one or both of the last two points are outside of the scissoring window that additional calculation takes place. If only one point is outside the window, a subroutine internal to the scissoring module is called with an indication that exactly one intersection with the window is to be found. If both points are outside the window, the subroutine is called with an indication that it should attempt to find two intersections.

Another service provided by the scissoring module is the removal of redundant blank lines. If a programmer inserts multiple consecutive blanked vectors into a graphic element, they will be coalesced into a single blank vector by this module.

SECTION 7.7: THE LINE STRUCTURE GENERATOR

This module, UGX102, is called by the device-dependent code to retrieve a line segment end point from a graphic element. Each time this module is called, it returns a single point with its blanking bit to the calling module, or it returns a signal which indicates that no more points are available.

The points in the graphic element are in the programmer coordinate system, but the points returned by this module are in the device coordinate system. The order in which the various operations are performed is:

1. A point is selected from the graphic element.
2. The point is transformed from the programmer coordinate system to the device coordinate system.
3. The point is passed to the scissoring module.
4. A point is requested from the scissoring module. If the scissoring module does not return a point, execution is transferred back to Step 1; otherwise, execution continues with the next step.
5. Lines returned by the scissor are returned to the calling module if solid lines are being drawn, otherwise the line is broken down into the requested line structure.

When line structure is being generated, the module keeps track of what it has done the last time it was called, and where it is on the line. For example, suppose dot-dashed lines are being drawn and the last operation was the plotting of a point. In this case, the next call to this module would cause it to move along the line the required distance (0.25 centimeters) and compute a point. This point is returned with a blank blanking-bit.

The remainder of this section gives some details on how a line is broken up to generate the line structure. A line consists of a number of straight line segments concatenated together. This line is parameterized so that the parameter is accumulated chord length along the line, measured in centimeters. Thus, the problem of evaluating the next point on the line is very simple after a line segment has been parameterized. Suppose the scissoring module returns a point (X2,Y2), and also suppose that the previous point was (X1,Y1) and that this point has already been assigned a parameter value of T1. The problem is to determine a parameter value T2 to be assigned at (X2,Y2) which assures that intermediate points can be evaluated on the line in the form:

$$X = AX*t + BX \qquad (1)$$
$$Y = AY*t + BY$$

for $T1 \le t \le T2$. The parameter t must measure line length along the line in centimeters. We also need explicit formulae for AX, BX, AY, and BY.

To solve this problem, we will need two additional values: YPAC and RUCH. First, let RPAC be the ratio of raster units per centimeter in the vertical direction to raster units per centimeter in the horizontal direction. RPAC is supplied by the device-dependent code and usually has a value of 1.0, although the value for a device like the GEN-COM 3000 is 0.8. The first

value, YFAC, is the quantity that a raster distance in the Y direction must be multiplied by to obtain the equivalent raster distance in the X direction. YFAC is the reciprocal of XFAC. The other value, RUCM, is the number of raster units per centimeter in the horizontal direction. RUCM is computed from the original scaling and scissoring limits (DDASCLDO and DDASCLPO) and the screen size (DDADPRSZ) in the DDA. To determine T2, first let:

```
M = max[abs(X2-X1),YFAC*abs(Y2-Y1)]
m = min[abs(X2-X1),YFAC*abs(Y2-Y1)]
```

Thus, M and m measure raster unit differences in X and Y in terms of horizontal raster unit values. To compute the length of the line, DT, in these units, we have:

```
DT = sqrt(M**2+m**2) = M*[1+(m/M)**2]**0.5                    (2)
```

Since this operation will be done repeatedly, a square root should be avoided. Therefore, we expand Equation (2) using the Binomial Theorem to get:

```
DT = M[1+(1/2)*(m/M)**2-(1/8)*(m/M)**4+...]                   (3)
```

Using the first two terms of Equation (3) gives the approximation:

```
DT = M + m**2/2*M                                             (4)
```

Equation (4) is clearly more easily evaluated than Equation (2) and is actually quite accurate. Equation (4) is exact when m=0, that is, for horizontal or vertical lines. The maximum error occurs at m/M=1, that is, for 45 degree lines. For m/M=1, Equation (2) gives M*1.414... while Equation (4) gives M*1.5. Thus, Equation (4) is always accurate to about 7%. This accuracy is quite good enough for its intended purpose. Thus, the value of T2 is:

```
T2 = (DT/RUCM) + T1
```

Finally, the equation of the line segment may be put in a form similar to Equation (1) by writing:

```
X = [(X2-X1)*(t-T1)/(T2-T1)] + X1
Y = [(Y2-Y1)*(t-T1)/(T2-T1)] + Y1
```

SECTION 7.8 THE CHARACTER STROKE GENERATORS

There are three character stroke generators available at present: UGCHAR1 for the BASIC character set, UGCHAR2 for the EXTENDED character set, and UGCHAR3 for the DUPLEX character set. These modules reside in the execution time library and are loaded as needed by the module UGX103. These modules are very similar to the line structure generator; each call to a stroke generator results in a single, scissored line segment being returned to the calling program.

The characters are defined within the stroke generators by a series of tables. The entries within these tables are generated by macros which are defined in the stroke generator itself. The first of these tables is the Character Table. For UGCHAR1, this table contains an entry for each defined character. The entry consists of the hexadecimal character code and an offset into the Primary Stroke Table. For UGCHAR2 and UGCHAR3, the table

contains an entry for all 256 character codes from X'00' to X'FF'. The second table is the Primary Stroke Table and consists of the actual definitions of the characters. The table entries consist of a series of blanking bits and Delta-X, Delta-Y movements. For UGCHAR1 and UGCHAR2, the Delta-X, Delta-Y movements are given by an offset into the Secondary Stroke Table; for UGCHAR3, the movements are given directly in the Primary Stroke Table. For UGCHAR1 and UGCHAR2, a third table, the Secondary Stroke Table, contains the actual Delta-X, Delta-Y movements.

The characters are defined on an internal raster grid. For UGCHAR1 and UGCHAR2, this grid has a spacing of six raster units between the centers of adjacent characters. For UGCHAR3, this spacing is 24 internal raster units. The principal function of the character stroke generators is to take a character in its internal raster unit format and scale, rotate, and translate the character into the raster grid of the actual device.

The stages of processing that the character stroke generators go through are the following:

1. Each character is selected from the given character string and matched with the Character Table. The proper position in the Primary Stroke Generator is found and the Delta-X, Delta-Y motions are converted to points on the raster grid of the actual device. The resulting points and their associated blanking bits are saved in some internal arrays.
2. The points that have been calculated are passed to the scissoring module and the points returned by the scissoring module are returned to the calling program.

Thus, the output of the character stroke generators consist of fully scissored points in the device coordinate system.

This section gives additional information about the non-interactive graphic devices which are supported by the Unified Graphics System.

SECTION 8.1: THE CALCOMP DRUM PLOTTER (MODEL 1136)

The CALCOMP Drum Plotter, Model 1136, consists of two distinct units. The first unit is the mechanical plotting unit and contains the plotting drum, paper feed and take-up rolls, plotting pens, and the motors and solenoids to drive these devices. The second unit is CALCOMP's Model 915 Controller and this unit contains the tape drive and a small computer. The small computer reads information from the tape and translates the information into stepping orders and transmits these orders to the mechanical unit. Thus, the format of the data that must be written on the tape is determined by what the program in the small computer expects to read. The documentation supplied by CALCOMP on this tape format is totally inadequate. The information necessary to incorporate this device into the Unified Graphics System was obtained by studying dumps of tapes prepared by CALCOMP's software for the IBM 360/370 computers.

There is only one interesting problem associated with a drum plotter. That problem is the drawing of points with a ball-point pen. A common way to try to draw a point, which works with a liquid ink pen, is to position the pen on the paper and draw a small square. The reason this will not work for a ball-point pen can be seen in the left-hand part of Figure 8.1.1. Drawing a square causes the ball to roll back and forth, but, unless the sides of the square are very large, does not cause the ball to roll far enough to bring ink from the ink reservoir to the paper. In addition, drawing a square leaves the ball in exactly the same orientation at the end of the figure as it was in at the beginning. Thus, drawing a number of points in this manner only results in the bottom of the ball being wiped clean of ink. However, if the scheme shown in the right-hand side of Figure 8.1.1 is used, the results are quite different. In this scheme, the pen is positioned at the lower left of the square and moved to the right and then up. The pen is then raised, repositioned at the starting point, lowered, and moved up and then to the right. This results in the ball never rolling back and forth and a continuous supply of ink is brought from the reservoir to the paper. Although it does take longer to draw a point using this scheme, it is necessary if such things as scatter plots are to be drawn with a ball-point pen. In the Unified Graphics System, all points are drawn in this manner.

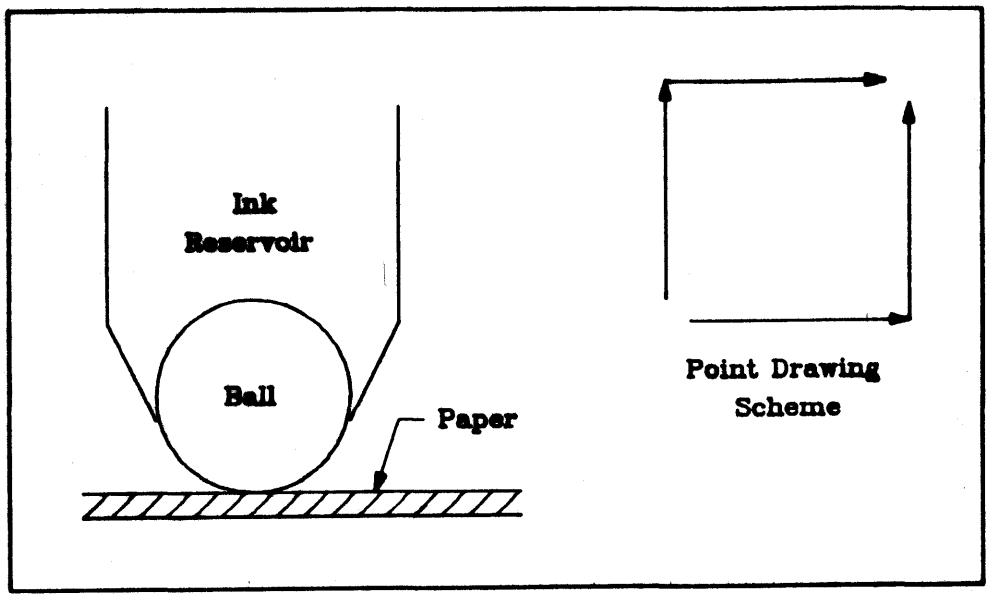

Figure 8.1.1: Drawing Points with a Ball-Point Pen.

SECTION 8.2: THE CALCOMP MICROFILM PLOTTER (MODEL 1675)

The CALCOMP Microfilm Plotter, Model 1675, also consists of two distinct units. The first unit contains a flat-faced CRT and the camera. The second unit is the same Model 915 Controller that is used in the Model 1136 Drum Plotter. The format of the data required by the program in the small computer is similar but more extensive than that required for the Drum Plotter. The documentation supplied by CALCOMP on this tape format is even more abysmally deficient.

SECTION 8.3: DISPLAY FILES FOR THE TEKTRONIX 4013

The TEKTRONIX 4013 Terminal is described in detail in [TEK-1]. At SLAC, these terminals are usually under the control of the WYLBUR Text Editing System [PAJ-1, WYL-1]. The pictures for this device are written into a card image partitioned data set, with each picture as a separate member. The records within each member are 80 characters in length, with at most 72 characters of information on each card.

The first card in each member consists of text identifying the member as a display file for the TEKTRONIX 4013, a clear screen command, and some idle characters. The GS (enter graph mode)

character is used as the idle character. Following the first card are additional cards containing 72 idle characters. The number of these cards depends on the value of the BAUDRATE options item. The picture itself begins after the idle cards and all orders are optimized as much as the BAUDRATE options item will permit. Each card containing pictorial information starts with a GS character and a four character beam positioning order. Following this on the card, are line and character plotting orders. All line plotting and beam positioning orders, except for the first one on each card, are optimized. Each card, except the last one, ends with an idle character. The last card ends with orders to re-position the beam to its home position, a US (enter text mode) order, and a series of five BELL orders.

The data that is put into the card images consists of EBCDIC characters and not the ASCII characters required by the terminal. The characters must be translated to ASCII, according to the following translate table, when they are transmitted to the terminal.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	--	--	--	--	--	--	7F	--	--	--	--	0C	--	--	--
1	--	--	--	--	--	--	--	--	--	--	--	--	--	1D	--	1F
2	--	--	--	--	--	--	--	1B	--	--	--	--	--	--	--	07
3	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
4	20	--	--	--	--	--	--	--	--	--	5B	2E	3C	28	2B	7C
5	26	--	--	--	--	--	--	--	--	--	21	24	2A	29	3B	7E
6	2D	2F	--	--	--	--	--	--	--	--	--	2C	25	5F	3E	3F
7	--	--	--	--	--	--	--	--	--	60	3A	23	40	27	3D	22
8	--	61	62	63	64	65	66	67	68	69	--	7B	--	--	--	--
9	--	6A	6B	6C	6D	6E	6F	70	71	72	--	7D	--	--	--	--
A	--	--	73	74	75	76	77	78	79	7A	--	--	--	5B	--	--
B	--	--	--	--	--	--	--	--	--	--	--	--	--	5D	--	--
C	--	41	42	43	44	45	46	47	48	49	--	--	--	--	--	--
D	--	4A	4B	4C	4D	4E	4F	50	51	52	--	--	--	--	--	--
E	5C	--	53	54	55	56	57	58	59	5A	--	--	--	--	--	--
F	30	31	32	33	34	35	36	37	38	39	--	--	--	--	--	--

Thus, the EBCDIC character X'C1' must be translated to the ASCII X'41'. EBCDIC characters marked by a pair of dashes will never be put into a card. The trailing blanks on the end of each card do not have to be transmitted to the terminal, although no problem will occur if they are transmitted. Since each card begins with an un-optimized beam positioning order, it does not matter if the transmitting program adds line feed or carriage return characters to the end of each card.

SECTION 8.4: DISPLAY FILES FOR THE GEN-COM 300-Q

The GEN-COM Terminal, Model 300-Q, is described in [GEN-1]. At SLAC, these terminals are usually under the control of the WYLBUR Text Editing System [PAJ-1, WYL-1]. The pictures for this device

are written into a card image partitioned data set with each picture as a separate member. The records within each member are 80 characters in length, with at most 72 characters of information on each card.

The first card in each member consists of mode select orders, text identifying the member as a display file for the GBN-COM 300-Q, and a carriage return. The picture itself starts on the second card. In the plotting mode, the terminal cannot keep up with the normal 300 baud rate; as a result, each card begins with a null character, is terminated with seven NULL characters, and NULL characters are interspersed at other positions in the cards. As a result, each card consists of approximately 15% NULL characters. The last card in a picture contains a BELL order.

The data that is put into the card images consists of EBCDIC characters and not the ASCII characters required by the terminal. The characters must be translated to ASCII according to the following translate table when they are transmitted to the terminal.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	--	--	--	--	--	--	7F	--	--	--	--	--	--	--	--
1	--	--	--	--	--	0D	--	--	--	--	--	--	1C	--	1E	--
2	--	--	--	--	--	--	--	1B	--	--	--	--	--	--	--	07
3	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
4	20	--	--	--	--	--	--	--	--	--	5B	2E	3C	28	2B	7C
5	26	--	--	--	--	--	--	--	--	--	21	24	2A	29	3B	7E
6	2D	2F	--	--	--	--	--	--	--	--	--	2C	25	5F	3E	3F
7	--	--	--	--	--	--	--	--	--	60	3A	23	40	27	3D	22
8	--	61	62	63	64	65	66	67	68	69	--	7B	--	--	--	--
9	--	6A	6B	6C	6D	6E	6F	70	71	72	--	7D	--	--	--	--
A	--	--	73	74	75	76	77	78	79	7A	--	--	--	5B	--	--
B	--	--	--	--	--	--	--	--	--	--	--	--	--	5D	--	--
C	--	41	42	43	44	45	46	47	48	49	--	--	--	--	--	--
D	--	4A	4B	4C	4D	4E	4F	50	51	52	--	--	--	--	--	--
E	5C	--	53	54	55	56	57	58	59	5A	--	--	--	--	--	--
F	30	31	32	33	34	35	36	37	38	39	--	--	--	--	--	--

The trailing blanks on the end of each card must <u>not</u> be transmitted to the terminal and no extra line feed or carriage return must be added to the end of each card. The transmitting program must <u>only</u> transmit the significant information on each card.

SECTION 8.5: THE VERSATEC ELECTROSTATIC PLOTTER (MODEL 1200A)

The VERSATEC Electrostatic Printer/Plotter, Model 1200A, is connected directly to the central computing facility. The special controller, supplied by VERSATEC, performs two functions: first, it simulates the essential functions of an IBM 1403 or IBM 3211 printer and its controller; second, the controller

recognizes special bit patterns within the print line which signal the controller to assume that the print line contains graphic data. This graphic data is essentially the X and Y coordinates of the end points of line segments. The controller converts this line data into the raster data required by the printer/plotter. The line segments are organized into "bands". The zero-th band is at the top of a fan-fold page and consists of LINEHX scan lines; the first band is below the zero-th band. Line segments must be ordered so that the controller receives all line segments starting in the I-th band before it receives the line segments starting in band (I+1). A complete description of the format of the graphic data will be found in [VER-1].

The device-dependent code for this device consists of five modules: UGVP121, UGVP122, UGVP123, UGVP124, and UGPPVEP. UGVP121 is the device-dependent open module, UGVP122 is the device-dependent code when the data is being sorted internally, UGVP123 is the device-dependent code when the data is being sorted externally, UGVP124 is the device-dependent close module, and UGPPVEP is a post-processor for the VERSATEC Electrostatic Plotter; it re-formats the sorted data when the data is being sorted externally.

The device-dependent code doing the internal sort works as follows: (1) the line segments for an entire picture are saved in blocks of length MAXMSA bytes. The data for each line segment includes its starting band number. (2) The lines segments in each block are sorted by band number, and finally, (3) the line segments in all of the blocks are merged together, re-formatted, and written to the output data set. Each line segment uses 10 bytes of data in one of these internal blocks.

The sorting method used is the "Quicksort" algorithm. Quicksort is described in detail in [KNU-1, RIC-1]. The specific algorithm used is described in Section 62 of [RIC-1]. This algorithm is a modified Quicksort which includes an improvement which speeds up the algorithm when equal keys are present. This enhancement is very important in this case because it is normal to have many line segments starting in the same band.

The device-dependent code which prepares data for the external sort is much simpler; it simply packs the line segment data into a buffer and writes it out when the buffer is full. This data is then sorted by the standard sort package, and the sorted data is re-formatted by UGPPVEP. The records written by the device-dependent code are 12 bytes long and consist of 6 half-words.

The first record is an initialization record and consists of:

HWD1: Zero.
HWD2: Not Used.
HWD3: Number of bytes per scan line.
HWD4: Number of scan lines per band.
HWD5: Zero for fan-fold paper, non-zero for continuous roll paper.
HWD6: Not Used.

Each of the remaining records describes a line segment. These records consist of:

- HWD1: Plot number.
- HWD2: Starting band number of line.
- HWD3: Relative X coordinate within the band of the start of the line segment.
- HWD4: Y coordinate of the start of the line segment.
- HWD5: Delta X value of the line segment. This value is always positive.
- HWD6: Y coordinate of the end of the line segment.

These records are sorted using HWD1 as a primary key and HWD2 as a secondary key.

The post-processor for the external sort data, UGPPVBP, may optionally accept a parameter string. The items NOOUT and DPRINT in this parameter string will be recognized. NOOUT suppresses all output to the SYSOUT data set. DPRINT causes the input data to be listed in the SYSPRINT data set. The principal use of these items has been in the check-out of the device-dependent code.

SECTION 8.6: A PARTITIONED DATA SET PSEUDO-DEVICE

The device-dependent code for this pseudo-device is quite straight forward. The only thing that is not completely obvious is the fact that the device-dependent code for both UGEPUT and UGEGET must be prepared to deal with graphic element arrays which are not stored in consecutive locations in memory. This difficulty cannot occur in FORTRAN, but in PL/1, two array can become interlaced if they are part of a structure. The device-dependent code checks for this possibility and obtains a temporary buffer using the GETMAIN macro when necessary.

SECTION 8.7: A SEQUENTIAL DATA SET PSEUDO-DEVICE

The device-dependent code for this pseudo-device is also very straightforward. The one thing that could use additional explanation is the format of the blocks of data in the output data set. This information will be necessary if the data is being transmitted to another computer. The output data set is a standard VBS data set, except that the full flexibility of a VBS data set is not utilized; records are, in fact, never spanned. A complete description of the VBS data format is found in [IBM-1], among other places. The format that is actually used is described below.

In addition to the data accessed by the user, a VBS data set also contains some control information. This control information consists of Block Descriptor Words (BDW) and Segment Descriptor

Words (SDW). Each block of data consists of a BDW followed by a number of segments.

The BDW consists of two 2-byte integers. The first integer is a count of the total number of bytes in the block (including the BDW). The second integer is zero.

Each segment consists of a SDW and one logical record of data.

The SDW also consists of two 2-byte integers. The first is a count of the total number of bytes in the segment (including the SDW). The second integer is zero.

The logical records, as described in the Unified Graphics System Programming Manual [BEA-1], are all an even number of bytes in length, except possibly for a text record (Type 5). These records are also forced to be an even number of bytes by padding the record with a null byte. The extra byte is counted in the record length count but not in the text character count. As a result of this padding byte, all SDW's begin on an even byte boundary and all data words are also correctly aligned in the block.

This section gives additional information about the interactive graphic devices which are supported by the Unified Graphics System.

SECTION 9.1: THE IBM 2250 DISPLAY CONSOLE

The IBM 2250 Display Console itself and the IBM 2840 Display Controller are described in [IBM-2]. The device-dependent code for the IBM 2250 uses the standard Graphics Access Method supplied by IBM. The Graphics Access Method is described in [IBM-3, IBM-4].

The only thing in the IBM 2250 device-dependent code that may be installation dependent is the way in which space in the IBM 2840 is allocated. At SLAC, the 16,384 bytes of memory in the IBM 2840 are partitioned at SYSGEN time into three nearly equal pieces with each piece being assigned to a different display console. Thus, in the Unified Graphics System, one invocation of the ASGNBFR macro is performed by UGOPEN to obtain one large block of memory in the IBM 2840. This block of memory is used throughout the execution of the job and is not released, by invoking the RLSEBFR macro, until UGCLOS is called. Some other programming packages for the IBM 2250 invoke the ASGNBFR macro each time a new display entity is added to a picture, and invoke RLSEBFR each time the memory space is no longer needed.

The organization of the display buffer is reasonably simple. The first two bytes are a Two-Byte-NOP. The second two bytes are a Start-Regeneration-Timer order. Next in the buffer are the graphic elements, followed by a terminal entry which consists of a JUMP back to the beginning of the display file. Following the terminal entry is a block of available space. The first four bytes of each element are either a Four-Byte-NOP (if the element is in the include state) or a JUMP to the next element (if the element is in the omit state). A keyboard input buffer always consists of (1) the NOP/JUMP control word, (2) an Enter-Vector-Mode order, (3) a blank vector to position the beam, (4) an Enter-Unprotected-Character-Mode order, and (5) the characters themselves. The device-dependent code keeps track of where the individual elements are located in the display buffer.

There is one options item that can be supplied to UGOPEN which is not described in the Unified Graphics Programming Manual [BEA-1]. This item, SYSDFIL, does the same thing as the options item MAXDFIL except that MAXDFIL is only allowed to ask for a maximum of 5376 bytes while SYSDFIL can ask for any amount.

SECTION 9.2: THE TEKTRONIX 4013 DISPLAY TERMINAL

The TEKTRONIX 4013 is described in [TEK-1]. In the interactive mode, the user's program communicates with the TEKTRONIX 4013 by passing records back and forth between itself and the HILTEN Terminal Manager. Communication is through the HILTEN Interface (HIF) written by Owen Saxton at SLAC. HIF is a set of FORTRAN or Assembler Language callable subroutines which control the writing to and reading from a terminal. The source for HIF is <u>not</u> included on the backup/distribution tape.

The device-dependent code for this terminal consists of the usual three modules (UGTK401, UGTK402, and UGTK403), in addition to a fourth module (UGTK404). This fourth module runs as a separate task and it is this task which does all of the I/O to the terminal through HIF. HIF, in fact, must be LINK-EDITED with UGTK404. The reason that a separate task is necessary is that HILTEN requires that a communicating program poll or wait for attentions from the terminal. This separate task does the polling in the Unified Graphics System and, when an attentions is received, creates an AQE and posts the attention ECB in the DDA.

The module UGTK402 writes to the terminal by generating records with a maximum length of 168 bytes and posting an ECB known to UGTK404. When this ECB is posted, UGTK404 transmits the record to the terminal and signals UGTK402 when transmission is complete. The records that are transmitted always start with an un-optimized beam positioning order and always end with an idle byte. The beam positioning and line drawing orders, except for the first one, are optimized as much as the BAUDRATE options item will permit.

The information in the records that are transmitted to HILTEN is in EBCDIC and not the ASCII required by the terminal. The Unified Graphics System assumes that HILTEN will translate the characters to ASCII according to the following table.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	--	--	--	--	--	--	7F	--	--	--	--	0C	--	--	--
1	--	--	--	--	--	--	--	--	--	--	--	--	--	1D	--	1F
2	--	--	--	--	--	--	--	1B	--	--	--	--	--	--	--	07
3	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	1A
4	20	--	--	--	--	--	--	--	--	--	5B	2E	3C	28	2B	7C
5	26	--	--	--	--	--	--	--	--	--	21	24	2A	29	3B	7E
6	2D	2F	--	--	--	--	--	--	--	--	--	2C	25	5F	3E	3F
7	--	--	--	--	--	--	--	--	--	--	60	3A	23	40	27	3D	22
8	--	61	62	63	64	65	66	67	68	69	--	7B	--	--	--	--
9	--	6A	6B	6C	6D	6E	6F	70	71	72	--	7D	--	--	--	--
A	--	--	73	74	75	76	77	78	79	7A	--	--	--	5B	--	--
B	--	--	--	--	--	--	--	--	--	--	--	--	--	5D	--	--
C	--	41	42	43	44	45	46	47	48	49	--	--	--	--	--	--
D	--	4A	4B	4C	4D	4E	4F	50	51	52	--	--	--	--	--	--
E	5C	--	53	54	55	56	57	58	59	5A	--	--	--	--	--	--
F	30	31	32	33	34	35	36	37	38	39	--	--	--	--	--	--

Since each record begins with an un-optimized beam positioning order, it does not matter if HILTEN adds blanks, line feeds, or carriage returns to the end of each record.

SECTION 9.3: THE SLAC EXPERIMENTAL DISPLAY CONSOLE

The user program communicates with the display console through the SLAC Real-Time Network [DEN-1, DEN-2, GRA-1, DEN-3]. The Real-Time Network transmits records between the central computing facility and the IBM SYSTEM/7 computers. The SYSTEM/7 computer transmits data to and from the display hardware.

The device-dependent code for this display console consists of four modules (UGSLTV1, UGSLTV2, UGSLTV3, and UGSLTV4). The fourth module runs as a separate task and it is this task which does all of the I/O to the display console through the Real-Time Network. The programmer callable entry points to the Real-Time Network must be LINK-EDITED with UGSLTV1 and UGSLTV4. The source for the Real-Time Network is not included on the backup/distribution tape. The task UGSLTV4 polls the Real-Time Network for an attention from the display console and creates an AQE and posts the attention ECB in the DDA when one occurs. The task also transmits any records created by UGSLTV2 to the Real-Time Network.

Records are sent, through the Real-Time Network, by the Unified Graphics System to the SYSTEM/7. The SYSTEM/7 also sends records to the Unified Graphics System. All of these records consist of a three-byte alphabetic identification and a colon, all in EBCDIC, followed by the actual data. Records sent from the central computing facility to the SYSTEM/7 are of four types, INT, DTA, BAC, and CTL:

A. Initialization Record: The data consists of a two-byte binary integer, giving the route number for all data which is to be sent from the SYSTEM/7 to the central computer.

INT:	Rtn.

No response is sent to the central computer for this record.

B. Data Record containing information to be added to the display file: The data consists of two two-byte integers giving the type (1 means normal element, 2 means keyboard input buffer) and the buffer location where the orders are to be inserted, followed by the display orders themselves.

DTA:	Type	Loc.	Display Orders

A response is always sent to the central computer when the data has been processed. The data transmitted in this manner always consists of either (1) a complete

graphic element, or (2) four-byte records which change the Omit/Include state of an element.

C. Data Read-Clear Request: The data consists of three two-byte integers giving the buffer location, length of an area in the buffer, and a flag. If the flag is negative, the area is cleared; if it is positive, the area is read; and if it is zero, the area is both read and cleared.

RAC:	Loc.	Len.	Flag

The response to the record contains any requested data. These requests always refer to entire graphic elements; partial elements are never manipulated by this record type.

D. Control Request: The data consists of a two-byte integer followed by other data whose format depends on the integer. The meaning of the integer is:
   1. Turn the display on.
   2. Turn the display off.
   3. Add the mouse cursor to the screen
   4. Read mouse cursor coordinates.
   5. Remove mouse cursor from the screen.
   6. Sound the audible alarm.

CTL:	Type	Additional Data

The response to the record contains any requested data.

Records sent from the SYSTEM/7 to the central computing facility are of two types, ACK and ASY:

A. Acknowledgement to DTA, RAC, or CTL: The data field is only present in certain cases, as described below:
   1. For a RAC with read, the data consists of two two-byte integers followed by the data in the buffer. The integers are zero if the keyboard cursor is not in this data. If the cursor is in the element, the first integer is the index of the word containing the cursor and the second integer is the index of the character within the word containing the cursor.
   2. For a CTL with mouse cursor read request, the data consists of two two-byte integers which contain the X and Y coordinates.

ACK:	Data

B. Asynchronous Data from SYSTEM/7: This data record is sent when the console operator has taken some action which must be communicated to the user program. The data consists of a two-byte integer followed by other data whose format depends on the first integer. The meaning of the first integer is:
   1. The ESCAPE key on the keyboard was pushed. No other data is present.
   2. A button on the mouse was pushed. A second two-byte integer gives the button number. A third

and fourth two-byte integer gives the X and Y coordinates of the mouse cursor.

3. A light pen attention has been generated. Two two-byte integers follow. The first integer is the address of the item being pointed to and, in the case of character data, the second integer is the offset of the specific character being pointed to.

ASY:	Type	Additional Data

The organization of the display buffer is quite simple. The first eight words are reserved for use by the SYSTEM/7 for such things as keyboard and mouse cursors. The rest of the buffer contains graphic elements. The first word of each graphic element is either a NOP (if the element is in the include state) or a JUMP to the next element (if the element is in the omit state). A keyboard input buffer always consists of (1) the NOP/JUMP control word, (2) positioning orders, and (3) character generating instructions. The device-dependent code running on the central computer keeps track of where the individual elements are located in the display buffer.

This section gives additional information about the non-standard graphic devices which are supported by the Unified Graphics System.

SECTION 10.1: THE VERSATEC PLOTTER AT THE 40" BUBBLE CHAMBER

The VERSATEC Electrostatic Printer/Plotter, Model 1100A, is used in an off-line mode. Tapes are written on the central computing facility and carried to the NOVA Minicomputer with the Model 1100A attached. The data on the tape consists of fixed-blocked records of 130 characters each.

The 130 character records consist of two parts, a two byte integer, and a 128 byte (1024 bits) string. If the integer part is zero, it indicates that a new picture is being started and a page eject should be sent to the Model 1100A. When the integer part of the record is zero, the 128 bytes do not contain any necessary information; actually, the bytes contain the EBCDIC characters for the job name and bin number padded on the right with blanks. If the integer part of the record is positive, it indicates that the 128 bytes form a string of raster data that is to be sent to the Model 1100A. A one bit in the string indicates a dark spot is required in that position. The string of bits must be repeated the number of times indicated by the integer.

There is one options item that can be supplied to UGOPEN which is not described in the Unified Graphics Programming Manual [BEA-1]. This item, SYSTEST, causes the device scaling and scissoring limits to be set to:

$$\begin{bmatrix} 0 & 119 \\ 0 & 119 \end{bmatrix}$$

The purpose of this item was to aid in the check-out of the device-dependent code. The first 120 bits (15 bytes) can be converted to a string of 120 characters and printed on the line printer. The printed output can be verified much quicker than obtaining actual plotted data.

SECTION 10.2: THE VERSATEC PLOTTER AT LASS

The users program transmits its graphic data to the plotter through the SLAC Real-Time Network [DEN-1, DEN-2, GRA-1, DEN-3] to the PDP-11/34. The records that are transmitted over the Real-Time Network all have a length of 406 bytes. The first four bytes contain the four EBCDIC characters 'UGD:' and the next two bytes contain a count of the number of bytes of data actually used in the rest of the record. The graphic data itself consists of up to 50 eight-byte blocks.

The eight-byte records are exactly the same as the eight-byte records defined by VERSATEC for their IBM 360/370 controller [VER-1] when it is used for graphics. Each picture starts in a new record and may consist of an arbitrary number of records. The first eight-byte block in a new picture is always an initialization record and signals that a page eject is to be given.

SECTION 10.3: THE TEKTRONIX 4013 DISPLAY TERMINAL

The device-dependent code for this device is very simple and straight forward.

SECTION 10.4: THE SLAC SLAVE SCOPES

The device-dependent code for this device is very simple and straight forward.

Ever since there have been two graphic devices of differing characteristics, programmers have talked about the problem of graphic device independence. Until the early 1970's, work on this problem had seldom passed the talking phase. In the early 1970's, a number of graphic subroutine packages, among them the Unified Graphics System, tried to solve some of these problems. This section will describe a few of these packages and will compare them to the Unified Graphics System.

SECTION 11.1: THE GRAPHICS COMPATIBILITY INTERFACE

GCI, the Graphics Compatibility Interface [GCI-1] was developed by Programming Methods Incorporated, of Palo Alto, California, under contract to the Ames Research Center of NASA at Moffett Field, California. This is one of the very few device-independent graphic packages that was in existence and publicly documented when the Unified Graphics System was started. GCI is coded principally in FORTRAN and was available on NASA's IBM 360/Model 67 computer.

GCI provides the user with a moderate number (about 40) of FORTRAN subroutines which may be used to generate pictures on a number of different devices. The subroutines make it reasonably easy to produce graphs with labels and titles. Although one of the early devices to be supported was an ADAGE, the system did not provide a means of writing interactive programs; any interaction had to be performed by device-dependent subroutines which were outside of GCI. A few of GCI's subroutines have arguments which are very similar to the OPTIONS argument in Unified Graphics subroutines. This idea was adopted from GCI and expanded in scope for inclusion in the Unified Graphics System.

SECTION 11.2: DISSPLA

DISSPLA, Display Integrated Software System and Plotting Language [DIS-1] is a proprietary product of Integrated Software Systems Corporation of San Diego, California. DISSPLA consists of a very large number (over 150) of FORTRAN subroutines to drive a wide variety of non-interactive graphic devices. With relative ease, a programmer can produce quite sophisticated plots and graphs with DISSPLA. Since the system is largely coded in FORTRAN, it is essentially independent of the host computer and is available on the IBM 360/370, CDC 6000/7000, and UNIVAC 1100.

One of DISSPLA's features is the ease with which new devices can be added. In DISSPLA, all plotting data is broken down into the operations of "pen up", "pen down", and "move pen", and these operations are passed to an interface module. The interface module then calls the device-dependent code which is usually supplied by the manufacturer of the graphic device. While this does mean that the addition of a new device is a simple matter, it also poses a problem; it is very difficult for DISSPLA to take advantage of the special features of a device. If the graphic device has a hardware character generator, for example, DISSPLA cannot easily use it. While DISSPLA has this narrow interface between the device-independent and device-dependent code, the Unified Graphics System has a much broader interface and can more easily utilize the special feature of a graphic device. The broad interface of the Unified Graphics System allows much greater flexibility, but does require more work when a new device is to be added.

SECTION 11.3: GINO-F

GINO-F, the Graphical Input and Output for FORTRAN System [WOO-1, WOO-2] was written at Cambridge University and at the Computer Aided Design Centre in Cambridge, England. Various versions of this system have been written, some principally in Assembler Language, and others principally in FORTRAN. GINO-F is widely distributed in England.

GINO-F incorporates an interesting and flexible scheme for processing the primative graphic elements. Suppose, for example, the device-independent code asks the device-dependent code to draw a dashed line. If the hardware can draw dashed lines, the device-dependent code draws the line and sends a "done" message back to the device-independent code; otherwise, it sends an "I-can't-do-it" message back. If the device-independent code receives an "I-can't-do-it" message, it responds by breaking the graphic primative down into simpler primatives, in this case the line would be broken down into the individual dashes.

This scheme makes it quite easy to support new graphic devices with extended hardware facilities. Suppose, for example, a graphic device with a hardware circle generator is to be added to a system like GINO-F. The software additions require that two modules be written; one which asks the device-dependent code to draw a circle, and another which breaks a circle down into line segments. In this way, circle drawing immediately becomes available to all graphic devices supported by the system, including those which do not have circle generators in their hardware. The Unified Graphics System's scheme of doing things would not allow such an addition to be put into the system in such a simple way. In the Unified Graphics System, the required changes would consist of (1) a module similar to UGEPNT and UGELIN, which allowed a programmer to define a circle, (2) a module similar to UGX101 and UGX102, which would retrieve a

circle from a graphic element in its original form or as line segments, and, unfortunately, (3) additions to all of the existing device-dependent code to call the circle retrieval module. The changes to the Unified Graphic System's device-dependent code would be very simple in each case but the large number of graphic devices involved means that a significant amount of work must be done.

SECTION 11.4: THE GRAPHICS COMPATIBILITY SYSTEM

GCS, the Graphics Compatibility System [GCS-1, GCS-2] was developed at the United States Military Academy at West Point, New York. GCS is largely coded in FORTRAN and has been distributed to a few other installations. GCS consists of a moderately large number (the primer describes 68) of subroutines which can drive a number of graphic devices including interactive devices.

GCS was one of the few early device-independent graphic systems which included interactive devices among the list of supported devices. GCS also uses character string arguments similar to the Unified Graphics System's OPTIONS argument to select non-default options for a graphic device.

SECTION 11.5: THE GRAPHIC DISPLAY SYSTEM

The Graphic Display System, also known as GD3 [MIL-1], was written at CERN in Geneva, Switzerland. The system runs on CERN's CDC 7600 and 6000 computers and supports both non-interactive and interactive graphic devices. GD3 is written principally in FORTRAN.

For non-interactive devices, a user's program does not directly generate orders for the graphic device. Instead, the actual output is a display file which contains a complete device-independent description of the picture. The GD3 system then provides a group of "interpreters" which can transform such a display file into the actual orders for any of the supported graphic devices.

The programming manual lists about 60 subroutines, but most applications will not need all of the subroutines. The number of arguments is small in most cases, so GD3 should be easy to learn and use.

SECTION 11.6: THE INTEGRATED GRAPHICS SYSTEM

The Integrated Graphics System [GOO-1, BLI-1] was developed at the University of Michigan in Ann Arbor, Michigan. This system is a late entry into the growing field of device-independent graphics packages. The Integrated Graphics System runs under the Michigan Terminal System on their Amdahl 470V/6 computer. The system has been distributed to a few other installations.

For simple pictures, there is almost a one-to-one correspondence between the Integrated Graphics System and the Unified Graphics System; however, certain areas of the Integrated Graphics System are more fully developed than the related areas in the Unified Graphics system. In the Integrated Graphics System, the creation of a picture is also a two-step process. The picture is first added to an internal data structure and, later, this data structure is converted to device-dependent orders and transmitted to the graphic device. In the Integrated Graphics System, the data structure is not directly accessible by the programmer, as a graphic element in the Unified Graphics System is, but this data structure has more versatility than the Unified Graphic System's graphic elements. The Integrated Graphics System's data structure contains a flexible transformation scheme and sub-pictures may be defined and utilized. The data structure may be either two-dimensional or three-dimensional. In the three-dimensional case, polyhedra may be defined and built-in hidden line removal algorithms may be applied to the polyhedra when the picture is generated.

The Integrated Graphics System seems to be fairly strongly coupled to the operating system. For example, the equivalent of the Unified Graphics System subroutines UGKPUT and UGKGET are performed by FORTRAN statements which write to unit 6 and read from unit 5, respectively. Also, one way to wait for a console operator to signal the application program to continue is by means of the PAUSE statement.

The Integrated Graphics System also tries to simulate light pens and screen pointers on interactive graphic devices which do not have them. Thus, the programming task is simplified because the programmer can always assume they are available. However, it is not clear how well this works because the simulation usually requires that the console operator type in the screen coordinates of the item or screen position to be selected.

In general, the Integrated Graphics System appears to be an interesting, well designed system. The designers of any future device-independent graphics systems should take some of the features of the Integrated Graphics System into consideration.

SECTION 11.7: THE GENERAL PURPOSE GRAPHICS SYSTEM

GPGS, the General Purpose Graphic System [GRO-1] was developed jointly by the Delft University of Technology and the Catholic University of Nijmegen both in the Netherlands, and the University of Cambridge in England. Two Assembler Language versions of the system were developed; one for an IBM 370 with a satellite PDP-11/45 and another for a stand-alone PDP-11/45. A third version, coded principally in FORTRAN, was developed at the University of Trondheim in Norway. Much of the design of GPGS was based on GINO-F. Development of the Assembler Language versions of the system began in 1972 while the FORTRAN version was started in 1974.

This system consists of a large number of subroutines, nearly 110. However, many of these subroutines will not normally be needed and many of the necessary subroutines fall into classes with very similar calling sequences. The result is a system that should be easy to use.

GPGS supports both non-interactive and interactive graphic devices. No attempt is made to simulate items like a light pen if it is not present in the hardware. Multiple graphic devices may be utilized in a single program; the selection of a "current" or active device is done exactly the same as in the Unified Graphics System. The data given to this system may be either two-dimensional or three-dimensional. A number of subroutines are provided to generate transformations of the three-dimensional data into the two-dimensional screen. There is, however, no provision for hidden line algorithms to work on the three-dimensional data structure.

SECTION 11.7: SOME ADDITIONAL COMMENTS

The Unified Graphics system has been used extensively at SLAC for a number of graphic applications. One of the things that has contributed to its acceptance is the fact that it is easy to learn and use. This ease of learning and use is due, in part, to the fact that only a few subroutines are necessary to begin programming. A core of only eight subroutines (UGBINT, UGBPNT, UGBLIN, UGBTXT, UGOPEN, UGCLOS, UGBPUT, and UGPICT) are necessary to utilize any of the non-interactive devices. Only six more subroutines (UGCTRL, UGBATN, UGDATN, UGRATN, UGKPUT, and UGKGET) are necessary to write programs for interactive devices. To make these subroutines easy to learn and use, a strong effort has been made to minimize the number of arguments in each subroutine.

The OPTIONS argument in most of the subroutines has contributed to the simplicity of the calling sequences. First, the options list has resulted in shorter argument lists because all of the device-dependent and seldom-used parameters are passed through the options list. Reasonable and consistent default options mean that simple options lists are all that is usually required.

Another result of the options list has been that most well written programs retain a certain amount of self-documentation. As a programmer begins to use fewer of the default options, the options lists become longer but the code retains, or even improves, in its readability. Some other systems have different schemes to handle such optional information. One approach is to add integer arguments to the subroutines which can take on a multitude of values. Such programs quickly become incomprehensible because of the difficulty in remembering what all of the different values can mean. Another common approach is to supply subroutines to "set modes" or have the programmer set values in a "mode array" which is later used by the system to determine character sizes, intensity levels, etc. In this way, the action of a subroutine is controlled, not only by its own arguments, but also by previous subroutine calls or program statements. These schemes are only trying to document subroutine side-effects and do not serve to make programs more easily understood.

The design of the Unified Graphics System has tried to put as many of the trade-offs between program size, efficiency, and speed under the control of the programmer. A simple instance is the size of the element array. A smaller array will save memory space but may cause the picture to be broken up into more pieces and this may increase execution time and probably will increase I/O accesses. The MAXENUM=<value> and MAXESIZ=<value> items associated with re-fresh interactive devices also represent places where the programmer can exercise control.

Finally, a few of the unusual features of the Unified Graphics System will be discussed. The Unified Graphics System is one of the few device-independent graphic systems to allow more than one device to be open at once. Most systems require that only a single device be used and this device is usually selected at LINK-EDIT time. Many interactive applications using the Unified Graphics System have found it convenient to also open a non-interactive device to generate hard copies of the pictures on the interactive device. Another place where the Unified Graphics System differs from most device-independent graphics packages is in its treatment of interactive devices. Most other packages, especially the earlier ones, either do not support interactive devices or support them through a late "add-on" to the system. Interactive devices were fully integrated into the Unified Graphics System since its beginning. A third area where the Unified Graphic System is different is in the number of programming languages which are supported. The vast majority are strictly FORTRAN packages. Some systems are claimed to support more than one language but, in fact, what has been done is to prepare a FORTRAN version and then call these subroutines from other languages, such as PL/1. The PL/1 Optimizing Compiler does provide a means of calling FORTRAN subroutines, but this can be very wasteful of machine time if done repeatedly. The PL/1 F-Level Compiler has no such feature and calling FORTRAN subroutines from it can force the programmer into extremely awkward constructions to pass character string and array arguments. Calling FORTRAN subroutines from PL/1 also can cause the I/O packages from both languages to become part of the user's

load module, and can cause serious difficulty in the PL/1 error processor. The Unified Graphics System fully supports both FORTRAN and the two PL/1 compilers; the PL/1 versions, for example, make full use of the dynamic storage allocation facilities inherent in the PL/1 environment.

An area where the Unified Graphics System is different from most other packages is that of scaling the picture to fit on the graphic device. Systems that process three-dimensional data are, necessarily, more complicated in this area than the Unified Graphics System. Among the two-dimensional systems, or the three-dimensional systems restricted to two-dimensions, the Unified Graphics System seems to offer more control over the aspect ratio of the picture. The Unified Graphics System can, in a completely device-independent manner, generate rectangular pictures of arbitrary size on a drum plotter; the same program can produce reasonable results on other types of graphic devices. Some systems allow only square pictures or arbitrarily clip off parts of the picture depending on the current graphic device.

This section contains a list of all of the publications that have been referenced in this document.

[BAR-1] J. Barlow and B. Franek, Graphic Representation of Functions of Two Variables, Rutherford High Energy Laboratory, Chilton England, Report No. RHEL/R 259 (August 1972).

[BEA-1] R. C. Beach, The SLAC Unified Graphics System, Programming Manual, Stanford Linear Accelerator Center, Stanford California 94305, CGTM No. 170 (February 1976, Revised December 1976 and January 1978).

[BEA-2] R. C. Beach, An Introduction to the SLAC Unified Graphics System, Stanford Linear Accelerator Center, Stanford California 94305, CGTM No. 166 (April 1975). This document is obsolete and has been replaced by CGTM No. 170.

[BEA-3] R. C. Beach, The SLAC Unified Graphics System, PL/1 Version, Stanford Linear Accelerator Center, Stanford California 94305, CGTM No. 142 (January 1973, Revised June 1973, November 1973, and September 1974). This document is obsolete and has been replaced by CGTM No. 170.

[BEA-4] R. C. Beach, The SLAC Unified Graphics System, FORTRAN Version, Stanford Linear Accelerator Center, Stanford California 94305, CGTM No. 143 (January 1973, Revised June 1973, November 1973, and September 1974). This document is obsolete and has been replaced by CGTM No. 170.

[BEA-5] R. C. Beach, The Internal Operation of the SLAC Unified Graphics System, Stanford Linear Accelerator Center, Stanford California 94305, CGTM No. 163 (April 1975). This document is obsolete and has been replaced by CGTM No. 171.

[BEA-6] R. C. Beach, A Version of the SLAC Unified Graphics System Coded Almost Entirely in FORTRAN, Stanford Linear Accelerator Center, Stanford California 94305, CGTM No. 191, (January 1978).

[BER-1] G. M. Berns, Description of FORMAT, a Text-Processing Program, Communications of the Association for Computing Machinery, Vol. 12, No. 3 (March 1969), pp. 141-146.

[BLI-1] J. F. Blinn and A. C. Goodrich, The Internal Design of the IG Routines, an Interactive Graphics System for a Large Timesharing Environment, Computer Graphics, Vol. 10, No. 2 (Summer 1976), pp. 229-234.

[CHA-1] H. Challenger, SHFNTB Macro, SLAC Computing Services, Stanford Center for Information Processing, Stanford California 94305, Technical Memo No. 88 (September 1974).

[COT-1] G. Cottafava and G. Le Moli, Automatic Contour Map, Communications of the Association for Computing Machinery, Vol. 12, No. 7 (July 1969), pp. 386-391.

[DEN-1] I. Denecke, Introduction to SLAC's Real-Time System, Stanford Linear Accelerator Center, Stanford California 94305, Real-Time Note 1 (September 1974).

[DEN-2] I. Denecke and C. Granieri, Message Routing in SLAC's Real-Time System, Stanford Linear Accelerator Center, Stanford California 94305, Real-Time Note 2 (December 1974).

[DEN-3] I. Denecke, L. Hundley, and J. Wells, The Role of the SYSTEM/7 in SLAC's Real-Time System, Stanford Linear Accelerator Center, Stanford California 94305, Real-Time Note 4 (April 1975).

[DIS-1] DISSPLA-Display Integrated Software System and Plotting Language, Integrated Software Systems Corporation, San Diego California, Form No. BD3-507, (December 1974).

[DIX-1] W. J. Dixon and R. A. Kronmal, The Choice of Origin and Scale for Graphs, Journal of the Association for Computing Machinery, Vol. 12, No. 2 (April 1965), pp. 259-261.

[EHR-1] J. R. Ehrman and G. M. Berns, FORMAT, A Text Processing Program, Stanford Linear Accelerator Center, Stanford California 94305, SLAC Report No. 135 (July 1971).

[EHR-2] J. R. Ehrman, User Program Statistics Gathering, SLAC Computing Services, Stanford Center for Information Processing, Stanford California 94305, Technical Memo No. 102 (July 1976, Revised August 1977).

[FAJ-1] R. Fajman and J. Borgelt, WYLBUR: An Interactive Text Editing and Remote Job Entry System, Communications of the Association for Computing Machinery, Vol. 16, No. 5 (May 1973), pp. 314-322.

[GCI-1] Graphics Compatibility Interface Users Guide, National Aeronautics and Space Administration, Ames Research Center, Moffett Field California 94035, (March 1972).

[GCS-1] Primer on Computer Graphics Programming, United States Military Academy, West Point New York 10996, (March 1974).

[GCS-2] Graphics Compatibility System - Programmer Reference Manual, United States Military Academy, West Point New York 10996, (1974).

[GBN-1] Model 300-0 Universal Data Terminal, Operators Manual, GBN-COM Systems Inc., New York 10017.

[GIA-1] T. Gianno, A Mathematical Method for the Automatic Scaling of a Function, Journal of the Association for Computing Machinery, Vol. 11, No. 1 (January 1964), pp. 79-83.

[GOO-1] A. C. Goodrich, Integrated Graphics System User's Guide, University of Michigan, Ann Arbor Michigan 48104, Computation Center Memo No. 299 (January 1976).

[GRA-1] C. Gramieri and J. Wells, The Command Language of SLAC's Real-Time System, Stanford Linear Accelerator Center, Stanford California 94305, Real-Time Note 3 (January 1974).

[GRO-1] D. Groot, E. Hermans, L. Caruthers, and J. Patberg, General Purpose Graphic System, Reference Manual, Rekencentrum, Technische Hogeschool Delft, The Netherlands, Publication No. RC-GPX-75002-A (November 1975)

[IBM-1] IBM System/360 Operating System, FORTRAN IV (G and H) Programming Guide, International Business Machines Corporation, Form No. GC28-6817.

[IBM-2] IBM System/360 Component Description, IBM 2250 Display Unit Model 2, IBM 2840 Display Control Model 1, International Business Machines Corporation, Form No. A27-2702.

[IBM-3] IBM System/360 Operating System, Graphic Programming Services for IBM 2250 Display Unit, International Business Machines Corporation, Form No. C27-6909.

[IBM-4] IBM System/360 Operating System, Graphics Access Method, International Business Machines Corporation, Form No. Y27-7113.

[KNU-1] D. E. Knuth, The Art of Computer Programming, Volume 3, Sorting and Searching, Addison Wesley Publishing Company, Reading Massachussetts, (1973).

[KUB-1] B. Kubert, J. Szabo, and S. Giulieri, The Perspective Representation of Functions of Two Variables, Journal of the Association for Computing Machinery, Vol. 15, No. 2 (April 1968), pp. 193-204.

[LEW-1] C. R. Lewart, Algorithm 463: Algorithms SCALE1, SCALE2, and SCALE3 for Determination of Scales on Computer Generated Plots, Communications of the Association for Computing Machinery, Vol. 16, No. 10 (October 1973), pp. 639-640.

[MIL-1] R. Miller, GR3, CERN, Geneva Switzerland, Computer Center Long Write-Up J510 (May 1976).

[RIC-1] R. P. Rich, Internal Sorting Methods Illustrated with PL/1 Programs, Prentice-Hall Inc., Englewood Cliffs New Jersey, (1972).

[TEK-1] TEKTRONIX 4013 Computer Display Terminal, Users Instruction Manual, TEKTRONIX Inc., Beaverton Oregon 97005, Document No. 070-1476-00 (1973).

[VER-1] Operating Manual, SC-360/370 Controller, Versatec Inc., Santa Clara California 95051, Part No. 70021-90001, (1976).

[WAT-1] S. L. Watkins, Algorithm 483: Masked Three-Dimensional Plot Program with Rotations, Communications of the Association for Computing Machinery, Vol. 17, No. 9 (September 1974).

[WIL-1] H. Williamson, Algorithm 420: Hidden-Line Plotting Program, Communications of the Association for Computing Machinery, Vol. 15, No. 2 (February 1972), pp. 100-103.

[WOO-1] P. A. Woodsford, GINO-Design and Implementation, University of Cambridge, Computer Aided Design Group, Cambridge England, Document No. 27.

[WOO-2] P. A. Woodsford, The Design and Implementation of the GINO 3D Graphics Software Package, Software - Practice and Experience, Vol. 1, No. 4 (October 1971), pp. 335-365.

[WRI-1] T. Wright, A Two-Space Solution to the Hidden Line Problem for Plotting Functions of Two Variables, National Center for Atmospheric Research, Boulder Colorado 80302, Report No. NCAR 72-26 (March 1972).

[WYL-1] WYLBUR/370 Reference Manual, Interactive Systems Group, Stanford Center for Information Processing, Stanford California 94305, (November 1975).