DEVELOPMENT OF AN INTELLIGENT INTERACTIVE GRAPHICS TERMINAL

ABSTRACT

An intelligent system, based on a refreshed display device and associated memory, which permits conversational dialogue with two host systems concurrently and has interactive graphics capability, is described. Its use, both as an intelligent terminal and a graphics device, is illustrated within the context of the Durham Education and Research Multi Access Network (DERMAN). Details of the communications aspects, graphics protocol and display file management are given. Future lines of development of the system are discussed.

P.S.Jones

University of Durham

1980

DEVELOPMENT OF AN INTELLIGENT INTERACTIVE GRAPHICS TERMINAL

A Thesis submitted for the
Degree of Master of Science
in the University of Durham

by

Paul Sheridan Jones

Department of Computing
University of Durham
July 198Ø

ACKNOWLEDGEMENTS

The project described here is part of a continuing development of facilities within the Durham Education and Research Multi Access Network. The original nucleus of software, which provided the basic graphical primitives from which the current system was developed *, originated from the Computing Centre, University of Michigan, USA, to whom due acknowledgment is given.

Two of the prime tests of a software system are its usability and adaptability to change. Many of the facilities encompassed by the current system have been established as a direct result of users' comments and suggestions. To them many thanks are due. Equally, the support of my colleague, G.R.Eadie, who provided the original 'springboard' for this development, was invaluable during the earlier part of this project. Finally, the advice and encouragement of my supervisor, Mr. J.S.Roper, is gratefully acknowledged.

---

* Aspects of the system relating to the main processing loop, local device commands and two session capability, were the author's original contribution.

CONTENTS

INTRODUCTION

1.1 General Introduction

This document describes the evolution of a system development facility used in the Durham Education and Research Multi Access Network (**DERMAN**). It is based on a refreshed display screen and associated memory, controlled by a dedicated processor. Some of its features were derived from developments in the 'intelligent' terminal field and from facilities available on more sophisticated devices at the Northumbrian Universities Multi Access Computer (**NUMAC**) central site (ref 1). Other elements in its makeup derive directly from the need to act as an interactive graphics display device. This document, although primarily a record of the project, is also intended as a part technical guide to the system. Consequently, machine reproducible diagrams and tables are interspersed with the body of text as appropriate.

The system runs on a PDP 11/10 central processor, and uses a VT11 display processor to display text and graphic information to the user. An operator's console and communications interfaces complete the specification. The two communications interfaces provide the route to the host systems selected. In this case these are usually a local UNIX system (ref 2), running on a PDP 11/34 with 128K words of store, and a remote IBM 370/168 at NUMAC, running the Michigan Terminal System (**MTS**) (ref 3). Terminal access to the IBM processor is achieved by a PDP 11/2Ø concentrator in Durham connected to Newcastle by a 9.6Kb share of a 48Kb synchronous line. These local PDP-11 processors, together with other minicomputers and microprocessors on the campus, represent the hardware of the DERMAN system. This facility, still in the early stages of development, will provide resource sharing, file transfer and other distributed computing facilities to machines on the University site.

```
      UNIX                     GTX
I----------I           I----------I
I          I           I          I
I   PDP    I           I   PDP    I       I--------I
I  11/34   I- - - - - -I  11/1Ø   I------I  VT11  I
I          I           I          I       I--------I
I----------I           I----------I
         |                |
         |                |
         |                |
         |                |
         |                |
   I--------------------I
   I                    I
   I                    I                       MTS
   I                    I                 I--------------I
   I     PDP 11/2Ø      I                 I              I
   I                    I. . . . . . . . .I IBM 37Ø/168  I
   I                    I                 I              I
   I--------------------I                 I--------------I
```

The Main Components of the DERMAN System

The software of the system, called **GTX**, allows the display screen to be shared by two independent terminal 'sessions' in normal text mode or to be completely utilised by one of them in graphics mode. Various device commands provide for adjusting the screen layout, local editing and printing functions. A feature of the system is the ability to have a type-ahead line, which is locally buffered. Users of the system found this extremely useful, particularly when using it to access MTS at NUMAC. This system operates in a half duplex mode and will not allow input from the user until prompted. With type-ahead, a complete line can be established locally and sent by typing a single carriage return character when the prompt is received.

The display tube incorporates a light pen, a device by which an item of character or graphical data displayed on the screen may be pointed at and recognised by the display processor. It is thus particularly useful for applications involving menu selection or picture manipulation. Earlier work with this equipment (ref 4) established its capability with an information system housed within the PDP 11/10 machine. With this current development, the 'driving' software for the graphics display resides remotely and talks to the GTX system by means of a simple protocol described later. GTX provides management of the display file structure, for example adding or deleting an object from the screen, and returns details of interaction by the user with the display as data to the host system. Such a system might be the Integrated Graphics (IG) (ref 5) package running in MTS, on which the protocol for the data exchange was based, or a graphical front end process to the INGRES (ref 6) data base management system operating under

UNIX.

The equipment on which the system is supported consists of

1) a PDP 11/10 processor with 16K words of memory
2) a VT11 display processor
3) a VR17 refreshed display unit with light pen
4) two RK05 disk units
5) two asynchronous serial line communications interfaces
6) a DECwriter hard copy console unit

In what follows, a certain level of familiarity with the characteristics of PDP-11 processors may be assumed at various stages. In particular the reader is assumed to understand the operation of a stack, the reference of devices as normal memory locations and the concept of an interrupt. A suitable reference which explains these and the more general features of this particular range of minicomputers is the manufacturer's handbook (ref 7).

1.2 Screen Layout

The layout of the screen at initialisation is pictured in fig. 1.1

```
****************************************************************
*                                                              *
*                                                              *
*                                                              *
*                                                              *
*                                                              *
*                                                              *
*                                                              *
*                                                              *
*                                                              *
*                                                              *
*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *
*                                                              *
*                                                              *
*                                                              *
*                                                              *
*                                                              *
*                                                              *
*                                                              *
*                                                              *
*                                                              *
*--------------------------------------------------------------*
*>                                                             *
*SYS P ATTN/C EOF/C FLOW/C FLO/H FLO/U COUNT NIBBLE VIS   LOCAL*
*  L - ØØØ177 ØØØØ3 ØØØØØ1 ØØØØØ ØØØØØ ØØØØØ ØØØ231 ØØØØØ ØØØØØ*
****************************************************************
```

Screen Layout
Fig 1.1

The dashed line in the centre of the screen divides it into two areas, LOWER and UPPER. These areas hold non-graphic character data received from the host systems and operate in a scrolling fashion. That is, when the area becomes full with text, the top line of text is removed and the whole image shifted up one line, allowing the next line to be inserted at the bottom of the area. The position of the line, controlling the division of the screen into the two areas, can be altered by a device command. Thus it is possible to have one system occupying just a few lines whilst the main area of the screen is available to the other. Characters from the keyboard are sent to the currently active host, which is also selectable by a device command.

In normal operation, keyboard characters are buffered in a type-ahead line at the bottom of the screen and can be edited. This line is prefixed by a prompt character (>) and a blinking underscore character (_) indicates the position at which the next character typed will appear in the buffer line. Editing of the line is performed by cursor control, allowing deletion or insertion of characters anywhere in the line. In addition certain system status information is displayed in the small area at the bottom of the screen.

In graphics mode the complete screen area is available for use. When expecting input in graphics mode, from the keyboard (for tracking cross positioning) or the light pen, the cursor character in the buffer line is changed to a plus sign '+' or a solid square character respectively.

1.3 Operation of the Display Unit

The VR17 display screen has a viewable area of size 1024 by 1024 raster units. A 6x8 dot matrix is used for the hardware generated character font and 8 hardware intensity levels are selectable. A full screen can display 42 lines of 73 characters each and the 96 ASCII character set, together with 31 special characters including Greek letters, is available. Full details of the configuration of the central processor, memory and display equipment, known as a GT40/GT42 in a packaged form, can be found in ref 8. The basic operation of this equipment will however be briefly described.

The central processing unit (CPU) and display processor both share the memory and essentially operate as autonomous devices. The display processor unit (DPU) executes a small set of display instructions and the display file through which it cycles is usually maintained as a linked list of display segments. The display file is maintained by a program executed by the central processor and the display processor is started by the CPU loading a value in the display program counter (DPC).

Once running, the DPU accesses memory by non-processor requests (NPR) and the CPU need not be aware of the existence of the DPU thereafter. However a means of communication between the DPU and the CPU is available by use of the 'display halt' (DHALT) instruction in the DPU. On executing this instruction the DPU will halt and send an interrupt to the CPU. Normally the interrupt service routine (ISR) executed by the CPU will restart the DPU within a short time and no visible change of the displayed image will occur. However the ISR can obtain details of which area or segment in the display file caused the interrupt and use this information to maintain an active display segment. This is usually performed by placing the DHALT at the end of a display segment and following it with the address of the next segment in the list. Thus the operation of this appears as a display jump (DJUMP) instruction to the next segment, with the advantage that the path of the DPU through the display file is monitored by the CPU.

The five basic instructions making up the display processor instruction set are

1) Set Graphic Mode
2) Jump
3) No-op
4) Load Status Register A
5) Load Status Register B

Instructions 2 and 3 are self-explanatory but the others require more amplification. The Set Graphic Mode instruction performs several functions. It primarily establishes the mode of the subsequent graphic data as one of the following :-

a) character mode
b) short vector mode
c) long vector mode
d) point mode
e) graph x mode
f) graph y mode
g) relative point mode

In character mode ASCII data will follow the instruction, two characters to a word. All vector mode data is relative, that is an offset from the current beam position, the distinction between short and long vector mode being that the increments are stored as two in a single word or as two separate words. Short vector mode allows a 6 bit co-ordinate increment, giving a maximum value of 63 units, whilst in long vector mode 1Ø bits, representing the full screen limit of 1Ø23, is available. Point mode, either absolute or relative, establishes a beam position. Relative mode uses a single word to store both the beam x and y increments and therefore each is restricted to a maximum of 63 units. Graph x/y modes are available for use in plotting data where either the x or y co-ordinate increments uniformly; this mode is never used by the GTX system.

Other functions contained within the Set Graphic Mode instruction are an intensity value in the range Ø to 7, light or dark vector, blink mode operation, line type selection and light pen interrupt enable. This latter function can be used to allow certain display segments to be sensitive to light pen interaction whilst others are not. That is, the programmer can decide which areas of the screen or items displayed are affected by the user pointing the light pen at them.

The Load Status Register A instruction contains bits to stop the display, select character font type and provide synchronisation with the line clock. This latter function may be used to attempt to maintain an image whose intensity is independent of the amount of data displayed. Since the DPU cycles through the display file, refreshing the display screen, it would normally produce a dimmer image as more data is added to the display file, since the refresh rate would decrease. By halting the DPU and restarting it on the next clock cycle, a 50Hz refresh rate, resulting in a constant intensity image, can be obtained. However if the cycle time of the DPU through the display file exceeds 20ms, then the display will appear to flicker, since the display will wait till the start of the next clock cycle, that is an interval of 40ms, before restarting. Thus choice of whether to employ synchronisation with the line frequency is not always obvious.

The Load Status Register B instruction is used for setting the graphplot increment when graph mode is in operation. It will not be mentioned further.

1.4 Software Aspects

The system is entirely written in MACRO-11, an assembler for PDP-11 machines running under the RT-11 operating system. However, the primitiveness of the assembler language is tempered by the addition of structured macros, which impose high level control functions on the underlying code. These include simple IF ..... ELSE and LOOP ..... REPEAT macros which generate test and branch type instructions. In more detail the macros available, arranged in groups , are :-

1) PUSH, POP, LOSE

2) CALL, RETURN

3) IF EQ (LT,LE,GT,GE) .... <ELSE> .... ENDIF

4) LOOP .... <BREAK (EQ,LT,LE,GT,GE)> .... REPEAT <(EQ,LT,LE,GT,GE)>

The group 1) macros provide for placing items on the stack (PUSH), taking items off the stack (POP) and dropping items from the stack (LOSE). A maximum of six arguments is allowed for each macro. They are typically used for saving registers on the stack as in

```
        PUSH    RØ,R2
```

and restoring them after use

```
        POP     R2,RØ
```

LOSE is used in a situation where the stack pointer is adjusted upwards by a number of words, for example

```
        LOSE    3
```

which will advance the stack pointer by 3 words.

The second pair of macros simply provides for a high level call/return interface for subroutine reference. Here they are simply replaced by the following instructions

```
          CALL    SUBR    ===>    JSR    PC,SUBR
          RETURN          ===>    RTS    PC
```

In a more general case these could be expanded to provide automatic register saving and restoration on subroutine entry and exit. In GTX most subroutines use only two or three work registers, so the convention adopted is that it is the responsibility of the called routine to preserve the integrity of any registers it uses.

Groups 3) and 4) are more substantial undertakings, given that nesting of these types of control functions is allowed. In essence what is required is a macro variable to hold the nesting level and another variable to store the generated label sequence counter. The first is incremented on entry to each IF clause and decremented on exit from each ENDIF macro. The second is incremented for each IF macro encountered. In the absence of an ELSE alternative, each IF <condition> statement generates a branch instruction for the complementary condition, to a label of the form QQn, where n is the label sequence value. In MACRO-11 facilities exist for using a symbol both as a numeric value and a character string of digits, as well as allowing concatenation of one string to another. At each ENDIF a label of the form QQn is output, to serve as target for the conditional branch of the corresponding IF statement.

With ELSE capability a little more care is needed. On encountering an ELSE, an unconditional branch is made to a label which will appear at the corresponding ENDIF and is of the form QZn where n is the label sequence. This is followed by output of a label QQn, which is the target for the corresponding conditional branch of the original IF statement. The nest level is then negated. All that remains now is that at ENDIF the nest level is tested. If positive a QQ label is output as before, but if negative a QZ label is produced. With an illustration the requirements may be a little clearer.

```
IF      EQ                                    BNE     QQ1
   ......                                     ......
   ......                                     ......
   IF   LE                                    BGT     QQ2
      ......                                  ......
      ......                                  ......
   ENDIF                        QQ2:
   ......                                     ......
   ......                                     ......
ELSE                                          BR      QZ1
   ......                       QQ1:          ......
   ......                                     ......
ENDIF                           QZ1:
```

Structured Macro - Instruction and Label Generation

A similar approach is used for the LOOP .... REPEAT construction which has the option of a BREAK statement in the loop body. An example of such a construction is shown here, where a search is being made of a linked list for a particular item. The list is terminated by a zero link.

```
MOV     HEAD,RØ            ; pointer to head of list
LOOP
   CMP  2(RØ),WANTED       ; is it the one we want ?
BREAK   EQ                 ; yes, get out
   MOV  (RØ),RØ            ; no, get next list element
REPEAT  NE                 ; and look out for end of list
```

Appendix A.1 contains a complete listing of the structured macros used.

The GTX system is divided into six modules, each separately assembled together with a module containing common symbol definitions and global declarations. These modules and their functions are as follows :-

GTMAIN	the main program. Performs the basic processing loop and handles communications aspects.
GTDEV	deals with device commands
GTSCRN	provides management for the 'nibble' pool and handles text characters for the scroll areas of the screen.
GTGRAF	provides the basic graphical primitives, handles the display file management including the light pen operation.
GTCI	character interpreter. Deals with all characters received from the host according to the IG protocol.

GTUTIL	assorted utility routines for internal to external conversion, character string output etc.

The object modules produced by assembling these source modules are then linked together using the LINK-11 linkage editor. The resulting image when loaded occupies approximately 4K bytes of memory. The commands necessary to generate the GTX system are given in Appendix A.3.

1.5 The Processing Loop

The main processing loop in GTMAIN is extremely simple and relies on a 'polling' of each device input queue to see if a character is available to be dealt with. By 'device' is meant the console keyboard or the host systems to which the system is connected. In essence its operation runs as follows. 'Examine each device queue in turn looking for a character to process. If there is none, continue with the next device immediately. Following the last device update the status display area, if necessary, and continue again with the first device'. Processing thus occurs on a single character per device basis rather than dealing with several characters for a device before proceeding with the next one.

Proceeding to look at this in a little more detail, a routine CHARIN deals with characters from the keyboard and a character interpreter (CI) module deals with characters from the host systems. The keyboard routine first looks for characters such as cursor control, interrupt, delete or tab characters which have special significance. Other characters are inserted in the type-ahead line until a carriage return character is received. At this point the characters are normally transmitted to the active host system, which is responsible for echoing of the characters. Short device command lines, which specify an attribute of the local system, are scanned for at this stage, that is on line termination.

The character interpreter module, which deals with characters from the host systems, functions in two basic modes, text and graphic. In text mode, characters received are displayed in a section of the screen set aside for that particular host and the system functions as a normal terminal. In graphics mode, characters received are interpreted according to the protocol used and normally display picture data using the full area of the screen. Character reception from the two hosts functions independently in text mode but not in graphics mode. In the latter case one host may send a control character which re-initialises the screen, thereby losing any text or graphics received from the other host.

The main functions involved in the processing are illustrated in fig 1.3. Some of the features shown there have not yet been mentioned; this diagram should be referred to as these items are discussed.

```
      INPUT                                               OUTPUT

I-----------I    I--------------I   I---------I
I keyboard  I--->I              I-->I device  I
I-----------I    I              I   Icommands I
                 I              I   I---------I
                 I   main       I
                 I processing   I
                 I   loop       I                I-----------I
                 I              I--------------->I to active I
                 I              I                I   host    I
                 I              I                I-----------I
I-----------I    I              I
Ihost line1 I--->I              I
I-----------I    I              I
                 I              I    I--------I  I-----------I
                 I              I    I char-  I  I   hard    I
                 I              I--->Iacter   I-->I   copy    I
I-----------I    I              I    I inter- I--|I-----------I
Ihost line2 I--->I              I    Ipreter  I-||
I-----------I    I              I    I--------I ||
                 I              I               ||I-----------I
                 I              I               |>I   text    I
                 I--------------I               | I  scroll   I
                                                | I-----------I
                                                |
                                                |
                                                | I-----------I
                                                ->I graphics  I
                                                  I-----------I
```

GTX Main Functions
Fig 1.3

MEMORY MANAGEMENT AND COMMUNICATIONS

2.1 Device Handling

The basic device components of the system, as shown in fig 2.1, are required for full utilisation of the systems capabilities.

```
   I-----------I        I---------------I
   I           I        I               I
   I   vr17    I        I  decwriter    I
   I refreshed I        I   console     I
   I  display  I        I   dl-11a      I
   I           I        I               I
   I-----------I        I---------------I
        II                    II
        II                    II
        II                    II
===========================================      UNIBUS
            II              II           II
            II              II           II
            II              II           II
   I-----------I  I-----------I  I-----------I
   I           I  I           I  I           I
   I 16K core  I  I  dl-11c   I  I  dl-11c   I
   I   memory  I  I interface I  I interface I
   I           I  I           I  I           I
   I-----------I  I-----------I  I-----------I
```

Basic System Components
Fig 2.1

The console, providing keyboard and hard copy printing functions, is connected to the UNIBUS by a DL-11A interface at a speed of 300bps. The two DL-11C interfaces, operating at a fixed speed of 2400bps, provide communication with the host systems required. The combination of the VR17 refreshed display screen and the core memory is often referred to as a 'GT40', this being the name given to a packaged variant of this equipment by the manufacturer.

Apart from the differing speed, the functional specifications of the communications interfaces are very similar (see ref 1Ø). The main difference between the DL-11A and the DL-11C is that the former is unable to indicate character framing errors on reception or to effect a break condition on transmission. Both the receiver and transmitter sections of the interfaces have a control and status register (CSR) and a data buffer register (DBR). These registers are addressed through the following locations :-

device	receive CSR	receive DBR	transmit CSR	transmit DBR
kl-11	17756Ø	177562	177564	177566
dl-11/1	17561Ø	175612	175614	175616
dl-11/2	17562Ø	175622	175624	175626

Device Register Addresses
Table 2.2

Within a device the relative positioning of the registers is identical; thus the only device dependent data required is the 'base address' of the device, taken to be the CSR. Full details of the specification and layout of the registers can be found in ref 1Ø; only those aspects deemed relevant to this application will be mentioned here.

At this point it may be useful to clarify the operation of a PDP-11 processor under the influence of an external interrupt, such as might occur from one of these devices. Two internal processor registers play an important part in this situation. The program counter (PC) addresses the next processor instruction to be executed. The processor status word (PSW) holds information about the current state of the processor; in the case of a PDP-11/2Ø the elements of the PSW are as follows :-

```
I5  ...bit....  7  6  5  4  3  2  1  Ø
I---------------I---------I-I-----------I
I  UNUSED       IPRIORITYI ICONDITION I
I               I         I I CODES     I
I---------------I---------I-I-----------I
```

The Processor Status Word (PSW)
Fig 2.3

The processor operates at one of eight levels of priority Ø-7. Devices of a hardware priority less than or equal to the current priority cannot interrupt the processor. Thus the priority scheme provides an effective interrupt mask.

The condition codes contain information on the result of the last CPU instruction executed and are set as follows :-

bit 3 (N) - set if the result was negative
bit 2 (Z) - set if the result was zero
bit 1 (V) - set if the instruction resulted in an arithmetic overflow
bit Ø (C) - set if the instruction resulted in a carry from the most significant bit

When an interrupt occurs the current PC and PSW values are pushed onto the stack, addressed through a general register, and new values for these registers taken from an area in memory specific to the device concerned.

This area is known as the interrupt vector for the device and the program module addressed through the new program counter is called the Interrupt Service Routine (ISR). Generally the value of the condition codes in the PSW loaded from the interrupt vector is not of direct consequence and can be used for other purposes, as we shall see. Normally upon completion of the ISR the reverse operation occurs, i.e. the saved values of the PSW and PC are restored from the stack, thus continuing the processor from the point at which it was interrupted.

Returning to the devices under consideration, their interrupt vector locations are as follows :-

device	receiver vector	transmitter vector
kl-11	60	64
dl-11/1	300	304
dl-11/2	310	314

Device Interrupt Vectors
Table 2.4

As before, the receiver interrupt vector location provides the device dependency.

The method used to provide common device support is now outlined. Since the devices are inherently similar, a common core of device support is clearly feasible; the question which naturally arises is, can the commonality be total? That is, can the devices share a single interrupt service routine ?

If the PSW is used to distinguish the devices, then a common ISR can indeed be achieved. Since the processor priority is normally set identically for devices of the same type, in this case priority 5, the condition code bits of the PSW are utilised to provide the device separation. They are set to Ø,1,2 for the respective devices. Thus the minimum information required to specify the devices is given in table 2.5 :-

device	interrupt location	base address	interrupt PSW
kl-11	6Ø	17756Ø	24Ø
dl-11/1	3ØØ	17561Ø	241
dl-11/2	31Ø	17562Ø	242

Communications Device Table
Table 2.5

This method can be used for a maximum of 16 devices of the same type. In GTX the above information is stored in a device table, together with an area for saving the original interrupt vectors.

The sequence of events following a receiver interrupt from one of these devices runs as follows. Firstly, the hardware saves on the stack the old program counter and processor status word and loads the new values from the interrupt vector for the device. In the interrupt service routine the new PSW is immediately pushed on the stack, followed by the saved registers used in the routine. Then the codition code bits of the stacked PSW are checked to ascertain which device caused the interrupt. This particular sequence of operations is required, since saving the registers (by a MOV instruction) sets the condition code bits of the PSW, thereby destroying those used to identify the device. The stack status during the processing of an interrupt is pictured below.

```
stack pointer                 stack

                            |        |
                            |        |
before interrupt ->         |        |
                            |--------|
                            |old pc  |
                            |--------|
after interrupt ->          |old psw|
                            |--------|
                            |new psw|
                            |--------|
                            |   rØ   |
                            |--------|
                            |   r1   |
                            |--------|
                            |   r2   |
                            |--------|
in interrupt routine ->     |   r3   |
                            |--------|
                            |        |
                            |--------|
```

Stack status during character interrupt routine

2.2 Memory Layout

The GTX system, although essentially a stand alone system not utilising any of the facilities of RT-11, co-resides in memory with the resident portion of the RT-11 operating system. In this way an 'escape' back into the local system is possible. For a fuller explanation of which parts of RT-11 are resident the reader is referred to reference 11. The layout of memory appears :-

```
         I======================I
         I                      I
         I resident part of     I
         I RT-11 operating      I
         I system               I
         I                      I
         I......................I
         I                      I
         I graphics 'nibble'    I
         I area                 I
         I                      I
         I                      I
~11ØØØ   I......................I
         I                      I
         I GTX program and      I
         I data area            I
         I                      I
 1ØØØ    I......................I
         I                      I
         I stack area           I
         I                      I
         I......................I
         I                      I
         I interrupt vectors    I
         I                      I
    Ø    I======================I
```

System Memory Layout
Figure 2.6

The GTX code and data area start at octal location 1ØØØ, immediately adjacent to the downward extending stack area. The area of memory between the top of GTX and the bottom of the RT-11 resident section is allocated to a list of graphics 'nibbles', each 64 bytes in size. This area is dynamically shared between graphics elements in use and a free pool of available elements. The number of free graphics nibbles is displayed in the status area at the bottom of the screen.

When the GTX system is started, an initialisation procedure is undertaken, which stores the interrupt vectors for the communications devices, as used by RT-11, in a save area and replaces them with those of its own. This allows an orderly return to be made to RT-11 at some later stage.

2.3 Nibble Management

The management of the nibble pool is probably the single most important part of the system, since it is used not only for all the graphical and textual information displayed on the screen but also for input/output buffering functions. Two primitives exist for obtaining and returning nibbles, performed by the routines NBLGET and NBLGIV respectively. Both of these routines update a free nibble queue head and a nibble element count. NBLGET returns a pointer to the new nibble obtained, whilst NBLGIV receives a possible chain of nibbles being returned to the free pool. NBLGIV takes two parameters, the head and tail of the unwanted chain. This is particularly useful when an entire graphics segment is being deleted, since only two instructions are involved.

Built around these primitives are two macros, NBLBYT and NBLWRD, used by the graphics functions to insert a character (byte) or a word into an open display segment. They require as parameters a pointer to the most recently inserted character (word) and a byte count of the space remaining in the current nibble. These updated values are then returned to the calling procedure, reflecting any changes, such as new nibble acquisition, when necessary. If the latter was required, then chaining of the new nibble to the display segment is automatically ensured.

2.4 Character Input and Output

On reception from the keyboard or the remote hosts, character data is collected by the interrupt service routine for later processing by the main program loop. The ISR makes calls to the PUTC routine to perform this function. In PUTC all buffering is performed dynamically, that is the buffer extent shrinks and grows as required. There is no limit on the buffer size allocated to a particular device, apart from the inherent memory limitations of the system. At initialisation no buffer space is allocated. On reception of the first character a 64 byte area is obtained from the nibble pool. Should the rate of arrival of characters temporarily exceed the processing speed of the main program loop, then further nibbles may be chained onto the initial element as required.

As characters are removed from the buffer, by calls to the GETCHAR routine, any disused nibbles are returned to the free pool. If the character queue empties completely, then the nibble containing the last character is likewise returned to the free list. When the queue is empty, calls to the GETCHAR character routine, to extract a character, return with the carry bit of the PSW (fig 2.3) set; if the carry bit is clear, it indicates that a character was successfully obtained. During extraction of characters from the queue by GETCHAR, the processor priority is raised to a level equal to that of the DL-11 devices. This ensures that character buffering and extraction are mutually exclusive operations. Two simple macros FREEZE and THAW, given in Appendix A.1, provide the tools for this mechanism, which is used generally in 'critical regions' of the system.

The device buffer, consisting of a linked list of nibbles, is accessed by a list head containing the character count and pointers to the first and last characters in the queue. This list head, in conjunction with a pointer to the device table, constitute all that is required to drive that particular device, and together are known as the device header. Fig 2.7 illustrates the structure of a device header. Since the nibble element size is 64 (octal 100) bytes, a simple 'address arithmetic' test, using a mask value of octal 77, is used to determine when a nibble boundary is being reached.

```
                              I-----------------------------I
                         ---->I  device address             I
                         |    I-----------------------------I
                         |    I   interrupt vector address  I
                         |    I-----------------------------I
                         |    I   new interrupt vectors     I
                         |    I-----------------------------I
                         |    I   old interrupt vectors     I
                         |    I-----------------------------I
                         |
                         |
                         |
I-------------------I    |
I device table ptr. I----|
I-------------------I
I character count   I
I-------------------I
I first pointer     I----|
I-------------------I    |
I last pointer      I-|  |
I-------------------I |  |
                      |  |
                      |  |    I----I
                      |  |    IlinkI---|
                      |  |    I----I   |
                      |  |    I    I   |
                      |  |--->I    I   |
                      |       I    I   |
                      |       I    I   |   character queue
                      |       I----I   |
                      |                |
                      |             I----I
                      |             IlinkI---|
                      |             I----I   |
                      |             I    I   |
                      |             I    I   |
                      |             I    I   |
                      |             I    I   |
                      |             I----I   |
                      |                      |
                      |                      |
                      |                      |
                      |                   I----I
                      |                   I  Ø I
                      |                   I----I
                      |                   I    I
                      |------------------>I    I
                                          I    I
                                          I    I
                                          I----I
```

Structure of Device Header
Fig 2.7

In the standard mode of operation the system is capable of receiving from both hosts concurrently, with little buffering of characters being required. That is, the main loop is able to extract and process characters faster than the rate at which characters are arriving. However if the console printer is being used as a hard copy device, as well as text appearing on the display screen, then flow control becomes an important factor. This is because the printer is limited to 3ØØbps whilst each remote line is rated at 24ØØbps. Flow control could be effected manually but this is tedious and almost impossible if a long listing is required. With this possibility some form of automatic flow control was decided upon, which would appear to the user as though no buffering was occurring. This control mechanism occurs irrespective of whether a hard copy listing is being produced, that is, it is operative all the time. In essence the character flow is controlled at two points, as follows.

Firstly, the arrival of characters from the host is governed by a low and high 'water mark', associated with the number of characters queued for that line. When the character count rises above a pre-determined 'high water mark', a flow control character is sent to the host, causing it to cease transmitting. When the level falls below a corresponding 'low water mark', another flow control character is sent, providing resumption of the paused output. This process is completely automatic and the user has no control over it.

Secondly, the extraction of characters from the device queue is governed by the state of an 'internal' switch, which is controlled by the user typing the flow control character on the keyboard. Thus when the user pauses the flow to the screen or printer, by changing this switch, it may well be that the 'external' flow from the host is still continuing (and vice-versa). When the internal flow is off, attempts to get a character from the device queue by GETCHAR always fail. The state of both flow switches is presented in the status display, for example

```
FLOW/H  FLOW/U
177777  ØØØØØØ
```

indicating that flow from the host is paused, whilst that to the user is on. Note that a single flow control character,set by a device command, is used for limiting both the external and internal flow of characters. Failure to set the correct flow character may lead to malfunction, since the system will eventually run out of buffer space.

With this buffering in operation, a little more care is needed in dealing with the interrupt character. As well as passing this character immediately to the host, which in the case of both UNIX and MTS has the effect of switching on character flow if it was off, it is also necessary to 'flush' the internal buffer of characters already stored.

Output to the hosts and keyboard is performed without interrupts. That is, when a character is to be output, the output routine PUTCHAR simply waits for the device to come to a ready state before transmitting the character. This is done by monitoring a bit in the device control and status register. When local character mode, described in Chapter 3, is in operation, PUTCHAR simply passes the character to PUTC directly, without addressing the device.

As characters are extracted from the keyboard buffer, they are displayed in the type-ahead line at the bottom of the screen. On reception of a carriage return character the line is output to the host system. If the line being typed fills the width of the screen, the full line is sent and the display line cleared ready for the rest of the line to be entered.

2.5 Display File Management

The display file through which the VT11 display processor cycles is constructed as a list of distinct segments or sub-pictures, each bearing an identity tag and other relevant information in a header block. Each display segment consists of at least one and possibly several nibble elements, the first of which contains the header information. Thus a typical display segment would appear :-

```
I----------I      I----------I       I----------I
I          I      I          I       I          I
I          I   ->>I          I    ->>I          I
I segment  I   |  I          I    |  I          I
I header   I   |  I          I    |  I          I
I          I   |  I          I    |  I          I
I..........I   |  I          I    |  I          I
I          I   |  I          I    |  I          I
I segment  I   |  I          I    |  I          I
I data     I   |  I          I    |  I          I
I          I   |  I          I    |  I          I
I          I   |  I          I    |  I          I
I          I   |  I          I    |  I          I
I----------I   |  I----------I    |  I----------I
I  DJUMP   I   |  I  DJUMP   I    |  I  DHALT   I
I----------I   |  I----------I    |  I----------I to next
I          I----  I          I----   I          I---->>
I----------I      I----------I       I----------I segment
```

Display Segment Structure
Fig 2.8

Each nibble in the segment except the last ends in a display JUMP instruction to the next nibble. The last nibble in the segment ends in a display HALT instruction followed by the address of the next segment header in the chain of display segments. This latter feature allows the system to maintain a 'currently active' display segment, since the display HALT instruction generates a processor interrupt on the PDP 11/10, as well as halting the VT11 display processor.

Knowledge of the active display segment is required for operations involving the light pen, for instance using the tracking cross or selecting a particular item from amongst those currently displayed. In addition when manipulations on a display segment are being performed, it may be necessary to wait until the display processor has left that active segment before making the adjustments. The five word segment header contains the following information :-

```
I-----------I
I  LASTY    I   - last y coordinate drawn in this segment
I-----------I
I  LASTX    I   - last x coordinate drawn in this segment
I-----------I
I CURR MODE I   - current mode  (text or vector)
I-----------I
I LAST ADDR I   - last used address in segment chain
I-----------I
IBYTE#I SEG#I   - bytes remaining in last nibble
I-----------I     + segment number
```

Display Segment Header
Table 2.9

Most of these are self explanatory; one or two require elaboration. LASTX and LASTY are stored so that if vector information is appended to the display segment, a choice of a short or long vector can be made, dependent on the current mode. LAST ADDR and BYTE# are used when inserting data into the graphics nibble by the nibble management macros.

All the graphics display segments are dynamically created and linked into the display chain, which contains the following permanent segments :-

1) a 50 cycle synchronising segment which causes the display processor to wait and restart with the next clock signal

2) a fixed status line at the bottom of the screen (fig 1.1) which contains fields giving

a) the currently active system and printer status

b) the attention character for the active system

c) the end-of-file character for the active system

d) the flow control character for the active system

e) the external flow status

f) the internal flow status

g) the count of characters in the type-ahead line

h) the number of free nibbles

i) the visible character mode indicator

j) the local character echo indicator

3) a keyboard type-ahead line of 72 characters

The upper and lower scrolling text segments are created early on in the initialisation process and re-created on subsequent 'clear screen' commands. They grow and shrink dynamically as required but have allocated at least one graphics nibble each. A fuller description of the operation of these segments is deferred till later.

At any instant there may exist an 'OPEN' display segment which is currently being created, appended or deleted. This is not to be confused with the ACTIVE display segment, which identifies the display segment the display processor is currently cycling through. The 'OPEN' segment is governed purely by the protocol of the characters sent to the system from the host. When a segment becomes the 'OPEN' segment, the segment header is copied to a set of open file variables, which are used in all subsequent references to the segment whilst it is open.

The primitives used in managing the display file are

CREATE	create a new display segment and make it the open segment
APPEND	add data to an already existing segment, leaving it as the open segment
DELETE	delete the segment from the display file
CLOSE	close the open segment. This involves copying the open file variables to the segment header.

Other auxiliary graphics functions are performed by the following routines :-

MOVE	move to point (x,y)
DRAW	draw to point (x,y)

VECTOR	make vector to point (x,y), deciding on short or long vector mode
TEXT	insert text character into open segment
FINDSEG	find segment header address given segment number
FINDNEXT	find next segment in display chain
FORCEMODE	force a display segment into desired mode
VRSTOP	interrupt routine for display halt instruction. Maintains active segment.
VRLPEN	interrupt routine for light pen interrupts. Controls tracking cross position.

DEVICE COMMANDS AND THEIR IMPLEMENTATION

3.1 Summary of Device Commands

Characteristics of the system are altered by the use of device commands. These include selecting the active host, setting tab positions and controlling the extents of the scroll areas. A device command is recognised by the appearance of the device command character, by default the escape character, at the beginning of a line. A summary of the device commands available to a user of the system is given below

Ac	set the attention (interrupt) character for the active system to c. This and two other 'immediate mode' characters are sent immediately when typed.
Dc	set the device command character to c .
Ec	set the end-of-file character for the active system to c.

Hn

set the inter scroll line height to n. This establishes the relative proportion of the screen allocated to the character areas. Initially 19, this parameter should lie within the range 1-37 inclusive.

I[U/L]

initialise display. If no parameter is given this removes all text from both scroll areas and any graphics displayed. If the parameter U or L is given, the text from that scroll area only is removed.

L

local mode switch. In local mode characters sent from the type-ahead line will be locally echoed to appear on the input stream, without being sent to a host.

Oc

set the output flow control character for the active system to c.

P[U/L]

printer switch. Is used to produce hard copy output on the console typewriter in addition to the refreshed display. The parameter specifies which host stream is to be copied; if no parameter is given then printing is disabled.

Rc

set the rubout (delete previous) character to 'c'. Typing this character will cause the character to the left of the cursor to be deleted.

S[U/L/R]

select active system to which keyboard characters are sent.

U - upper system

L - lower system

R - RT-11 (local) system

The system currently active is displayed in the statistics area at the bottom of the screen.

Tn1,n2,n3...

set input tab positions at columns n1,n2 ... etc, which are expanded with spaces on input. Up to twenty tab positions are allowed. Tab characters received beyond the largest tab setting given are replaced by a single space. Specifying T alone disables tab expansion and the tab character has no special significance.

U	set unbuffered input mode. Every character typed is forwarded to the host directly and not buffered or interpreted locally. The device command character still assumes significance however.
V	visible character mode switch. In visible mode all characters received are printed as ASCII octal codes on the console printer.

All alphabetic characters in device commands may be given in either upper or lower case. A line beginning with the device command character, but not interpretable as a device command, is sent to the host.

Some of these device commands are sufficiently explained by the above. However, a few of them will be discussed and elaborated in the following sections.

3.2 Screen Initialisation

The initialise screen command re-establishes the screen to its original layout with one exception, that the currently existing division between the two scroll areas is maintained. That is the dashed line shown in fig 1.1 is left at its current position. This is possible because the adjust scroll height command updates the scroll table information as well as the segment headers themselves. A fuller explanation of this appears in Chapter 4.

3.3 Scroll Area Adjustment

The relative amount of the screen taken up by the lower and upper scroll areas can be adjusted. At initialisation each scroll area is established by reference to a scroll table containing the following information

Attribute	Word
segment number	Ø
X coordinate	1
Y coordinate	2
maximum number of lines	3
current number of lines	4
number of characters in line	5
number of characters per line (width)	6

Scroll Table
Table 3.2

The X,Y coordinate pair define the position at which the first character to enter the scroll area will appear and are referred to only at screen initialisation. Reference to the other quantities is made as characters are received by the scroll processing routines, in order to effect scrolling and line wraparound. In addition a variable holds the value of the scroll area divider height, expressed as a line count. When the scroll adjustment device command is given, this value is directly updated and the maximum line count and the (X,Y) coordinates in the scroll table are changed. In addition one of the scroll areas is reduced in size and the origin of the upper scroll modified. This can best be seen by an example. Consider the screen at A) being changed to the screen at B) , where the lower scroll area is extended further up the screen.

```
I-----------------------I     I-----------------------I
I line u1               I     I line u5               I
I line u2               I     I line u6               I
I line u3               I     I - - - - - - - - - - - I
I line u4               I     I                       I
I line u5               I     I                       I
I line u6               I     I                       I
I- - - - - - - - - - - -I     I                       I
I line 11               I     I line 11               I
I line 12               I     I line 12               I
I-----------------------I     I-----------------------I

          A                             B
```

Fig 3.3 Scroll Height Adjustment

Firstly the maximum line counts in the scroll tables are modified by adding the appropriate value to the lower and subtracting from the upper. This amount is calculated by the difference between the newly specified line position and the curently stored value.

Then the upper scroll is 'shrunk' by sending it a null character. This is ignored but forces the scroll routine to re-examine the status of the current number of lines in the scroll against the now modified maximum line count. It then proceeds to reduce the area in size by repeated scrolling as described in Chapter 4. That is, the text now starts at a position several lines down from the top of the screen. What remains to be done is to move this start position back up, by adding an appropriate amount to the Y co-ordinate, thereby lifting the reduced area back up to the top of the screen. Apart from the line count modification, nothing further is required for the lower scroll, which will continue to expand upwards until the new line limit is reached, when scrolling will be re-activated.

An identical method is used for the opposite situation, that of an expanding upper and shrinking lower area, except that in this case the contents of the upper scroll area is shifted down the screen, so that the last line of text remains immediately above the division line.

3.4 System Selection

Selection of the active system, that is the system to which the characters from the type-ahead line will be transmitted, is simply implemented. Since a common routine, PUTCHAR, outputs a character to a device and expects two parameters (the character and the device header), it simply involves passing the active device header, which is stored in a variable ACTDEV. This variable is changed by the <esc>S device command and the currently active system is displayed in the status area as

```
SYS
--X
```

where X is L,U or R for lower,upper and RT-11 systems respectively. When a return to the RT-11 operating system is specified, the previously saved interrupt vectors for the devices are restored. Subsequently typing two control/c characters will effect an escape back to RT-11. When the active system is changed the 'immediate mode' characters (see 3.6) for the selected system are automatically displayed in the status line. If local mode is in operation, the active system establishes the apparent host from which the characters will be echoed. PUTCHAR tests for local mode and ,if specified, places the character on the input queue of the active device.

3.5 Character Editing and Cursor Control

As a normal character is typed on the keyboard, it is placed in the type-ahead line and the cursor, consisting of a blinking underscore character, is moved one position to the right to indicate the position at which the next character typed will be entered. When the rubout (delete previous) character is typed, the character to the immediate left of the cursor is removed from the screen and the cursor moved one position to the left. Thus successive rubout characters will delete a sequence of characters from right to left.

Without cursor control, typing errors may require the rubout of several characters to correct an error, followed by the retyping of the correct characters just deleted. With cursor control it is possible to move the cursor to any position on the line and then insert or delete an arbitrary number of characters, giving full line editing capability. The control characters used to position the cursor are control/l and control/r for left and right adjustment respectively.

One point bears mentioning here. Throughout the description it has been assumed that the input record terminator is a carriage return character and cannot be changed. Therefore it seems sensible to allow a carriage return character to flush the type ahead line even when the cursor is not at the extreme right of the line; that is, a carriage return character typed during insert or delete mode will cause the contents of the line, upto the rightmost character displayed, to be transmitted.

3.6 Immediate Mode Characters

Normally as characters are typed they are buffered in the type-ahead line and are not processed further until a carriage return character is received. Howevever there is usually a requirement that some characters are acted upon immediately they are typed, for instance to send an interrupt to the host. These characters are called immediate mode characters.

By means of device commands the host dependent immediate mode characters can be altered. For the MTS and UNIX systems to which the terminal is normally connected, the immediate mode characters and functions are as follows :-

function	UNIX	MTS
to interrupt the current task	break	break
to indicate an 'end of file' condition	cntrl/d	cntrl/c
to pause the output to the terminal (flow control)	cntrl/o	cntrl/a

Immediate Mode Characters
Table 3.1

Many systems use the 'break' key to signal an interrupt condition to the host (in MTS break is the default key for an attention interrupt, whilst in UNIX either break or delete can be used). However since the DL-11A interface is not capable of indicating when the break key on the console keyboard has been depressed, some other character must be used to indicate this function, if it is required to send a break to the remote system. In this case the delete character was chosen, so that attempting to transmit an ASCII <177> would cause the break condition to be generated on the DL-11C for a suitable period.

3.7 Unbuffered Input Mode

For those applications which require character at a time working, an unbuffered input mode can be specified. In this case every character typed is immediately sent to the host with one exception. If the device command character is typed, then the line is processed normally in buffered mode. This permits device commands to be scanned for, thus allowing, for example, a switch back to unbuffered mode to be made.

3.8 Input Tabbing

Adjustable tab positioning was felt to be very useful particularly when inputting assembly language , structured language source or data where a regular format or indentation is required. The implementation of tabbing is quite straightforward and is performed recursively by internal calls to the 'character in' routine with the space character as parameter.

3.9 Hard Copy Ability

Output to the printer of the console is performed using the same routines as output to the host systems, the adjustable parameter being the device header. When a character received from a host is processed by the GTCI routines, an additional call to PUTCHAR , with the console device header specified, is all that is required for hard copy printout. This occurs in addition to the normal display function. The print state is displayed in the status line as either U, L or - ; indicating that hard copy of the upper or lower system is occuring or that the hard copy function is disabled.

3.1Ø Local Character Echo Mode

The local mode command was originally installed as a debugging aid. It allows the system to function, at least in character mode, without the need to be connected to a host computer. What it does is to 'feed' the characters from the type-ahead line, which would normally be sent to a host, into its own input buffers, thus simulating an immediate character echo. This allowed debugging of the software responsible for screen area scrolling and adjustment to be performed in a controlled and reproducible manner. However, having seen it demonstrated, one user soon utilised this for establishing a 'ruler', consisting of a repeated sequence of the digits Ø to 9, in the lower scroll area; this can then be used to position information accurately or identify a particular column quickly.

3.11 Visible Character Mode

For debugging purposes and at other times when it is required to 'see' all the characters transmitted to the system from the host, a 'visible' character mode can be specified. In this mode all characters received are printed as ASCII octal codes on the console. This mode can be specified in addition to hard copy print, in which case by typing the string '1234' the following output would occur :-

```
1<Ø61>2<Ø62>3<Ø63>4<Ø64>
```

CHARACTER AND GRAPHICS MODE

4.1 Operation of the Scroll

The operation of the scroll areas and their interaction with the display file management will now be described. As stated in Chapter 2, each display segment has a segment header at the beginning of the first nibble of the segment. In the case of the scroll areas the rest of the first nibble is unused, except for the initial (X,Y) coordinates of the first text character. That is, the text for the scroll always starts at the beginning of the second nibble in the segment. This facilitates the implementation of the scrolling mechanism, as we shall see. The body of the text for the scroll segments will generally contain line feed (LF) characters and extend over several nibbles as shown in fig 4.1

```
I----------I      I----------I      I----------I
I          I   ->>I          I   ->>I          I
I segment  I   |  I          I   |  I     ! LF I
I header   I   |  I LF !     I   |  I          I
I          I   |  I          I   |  I          I
I..........I   |  I          I   |  I          I
I  APNT    I   |  I          I   |  I          I
I    X     I   |  I     ! LF I   |  I          I
I    Y     I   |  I          I   |  I  LF !    I
I TEXT     I   |  I          I   |  I          I
I.. null ..I   |  I LF !     I   |  I          I
I.. null ..I   |  I          I   |  I     ! LF I
I----------I   |  I----------I   |  I----------I
I  DJUMP   I   |  I  DJUMP   I   |  I  DHALT   I
I----------I   |  I----------I   |  I----------I to next
I          I----  I          I----  I          I---->>
I----------I      I----------I      I----------I segment
```

Scroll Segment Structure
Fig 4.1

The APNT point mode instruction in the first nibble establishes the starting position of the text of the scroll. The TEXT 'set character mode' instruction defines all data following as character data. In character mode, the null character (ASCII zero <ØØØ>) produces no effect on the screen and can be viewed as a text mode no operation (NOP). The rest of the first nibble, up to the display jump instruction, is filled by such null characters. This is a by product of the initial CREATE call made to establish the scroll segment, which zeros the nibble as it draws it from the free pool. Thus the first nibble contains no displayable text data.

As character data is received by the scroll routine, it simply adds it to the next free space in the last nibble, chaining in a new nibble as required. When a line feed character is received, the current character count in the scroll header is set to zero and incremented whenever a character other than LF is received. Should the character count exceed the maximum stored in the scroll header, the routine provides for software 'wraparound' by generating an internal line feed/carriage return sequence. Without this feature, long lines of text would be truncated at 72 characters.

As a line feed character is received, the Y coordinate stored in the first nibble is updated and the current line count is incremented. The latter is then compared with the maximum in the scroll header. If it is exceeded, a scrolling operation is performed as follows. The second nibble of the segment is accessed and characters are set to null (zero) until the first line feed is encountered. This character is then also cleared, and the Y coordinate and line counts decremented. If, in searching for a line feed, the end of the nibble is reached, it is returned to the free pool and the reference from the first nibble updated to point to the next nibble in the segment.

In normal operation, as one line enters at the bottom of the area, one line disappears off the top. However, when the scroll area is being adjusted to a reduced size by a device command, this operation is repeated several times, removing lines from the top of the area. In this special situation, a null character is passed to the scroll routine, which is not displayed but merely forces the line count check and subsequent scrolling operations to be performed.

When a character is passed to the scroll routine, an initial check is made to see if the scroll segment is the currently open display segment. Since characters output from the two host systems may be interleaved, it may be necessary to close the previously open scroll segment associated with one host and open the scroll segment of the other. In the situation of maximum interleaving, where alternate characters come from the same host, this closure and opening operation will occur for each character. However, in a more likely situation, where a burst of characters from one host may be followed by a group from the other host, each scroll segment will remain open for several characters at a time.

If the scroll is not already open, a call to APPEND is made, which will close the currently open display segment and open the specified one. Opening the segment essentially copies the segment header to a set of open file variables, of which the last used nibble address is particularly relevant. The latter defines the position of the next free space in the last nibble of the segment. As a character is received, the scroll routine inserts it at this position by calls to the nibble management macro NBLBYT, which will chain in a new nibble if required.

Clearing a scroll area of all displayed text is achieved by passing the scroll routine a special character (ASCII <Ø25>), or by calling an entry point within the routine which has the same effect. This makes a CREATE call for the segment, using the (X,Y) values currently in the header for the scroll origin. When CLOSE is called, it will automatically delete a segment of the same name as that currently OPEN, thereby destroying any old version of the scroll.

4.2 Graphics Protocol

Since GTX allows the system to function as a simple terminal, the protocol it uses for controlling the graphics primitives is based on a small group of control characters at the lower end of the ASCII character set. In normal operation, the character interpreter (CI), which deals with characters received from the hosts, watches for these control characters and branches to an appropriate section to perform the required action. If the character is not a control, the CI is said to be functioning in **terminal mode** and the outcome is simply to pass the character to the text scroll routine.

Reception of certain control characters may cause the CI simply to perform an action, such as clearing the screen, and revert immediately to terminal mode. Whilst the action is being performed, the CI can be thought of as existing in a transitory state, for example 'clear screen state'.

Other control characters may have the same basic effect, except that the duration of the state is dependent on external events, such as the user typing a character on the keyboard. Nevertheless, the next character processed by the CI will be seen in terminal mode.

There also exists the situation where a control character can switch the CI to a different state, in which it continues to process received characters, before eventually returning to terminal mode. It may even exist in various sub-states within the main state. In this situation, a mechanism must be established to allow the next character received by the CI to be directed to the correct state section, since the main processing loop sees the CI as a single state 'black box'. Before going on to look at the specific control characters used in this system and the states they induce, it is pertinent to examine how the CI deals with control characters and the state switching mechanism.

Once a character has been processed and the CI wishes to prepare for receiving the next character, it does so by providing a single exit path from the routine for all states. This exit path is called as a subroutine, using the program counter as the link register, with the result that the address of the instruction following the subroutine call is placed on the processor stack. The exit routine 'pops' this address to a safe place and itself does a normal subroutine return, thereby returning to its caller's caller, that is the main processing loop.

On the next entry to the CI, it simply uses the saved return address in a jump instruction to ensure the character is received by the appropriate state section. In this way a 'stepped' access to the various parts of the module, according to the underlying protocol, is achieved.

One specific assumption made within the CI is that all sections, which call 'exit' to get the next received character, do so on the understanding that any control character (less than ASCII <Ø4Ø>), will arrive at the saved return address, whilst normal characters will appear at one word past this location. For example

```
..........

..........

CALL  EXIT     ; exit to get  next character

BR  CON        ; control characters will come here

..........     ; and normal characters here

..........
```

The five 'non terminal' states identifiable within the character interpreter are as follows :-

1) Initialise (clear) the screen
2) Transmit status response (answer back)
3) Enable light pen interaction
4) Enable tracking cross positioning
5) Enter data into display segment

The first four states are triggered by reception of a single character as follows :-

1) ASCII FF (form feed) <Ø14>

2) ASCII ENQ (enquiry) <ØØ5>

3) ASCII DC2 <Ø22>

4) ASCII DC4 <Ø24>

The fifth state, that of entering data into a display segment, is achieved by a sequence of control characters. The form of this is

<soh><segment no.><stx><..... assumed TEXT data .....>

where <soh> is the ASCII 'start of header' character and <stx> is the 'start of text' character. The value of the segment number transmitted is arranged to be a printing (i.e. non-control) character by the addition of a suitable constant. This is a general feature of the data contained in the dialogue between IG and the GT4Ø, where only a very small set of control characters are legitimate and are used to change the state of the character interpreter.

Initially, the data following the <stx> is assumed to be text data. To switch to vector mode an ASCII SO <Ø16> character is inserted and followed by the vector data, suitably transformed to be non-control. To revert back to text mode, a leading SI <Ø17> character is used in the same way. Any other control character is considered invalid and the state is restored to terminal mode. A formal definition of the display segment protocol is given in Apendix A.2.

4.3 Text and Vector Processing

Text characters, whether destined for the scroll area of the screen or as part of a graphics display segment, are processed by the TEXT routine in the GTGRAF module. One of the first things it does is to test if the current mode is already text. If not, it must first insert a character 'set graphic mode' instruction into the nibble and update the current mode indicator in the open file variables. Then the character is inserted into the nibble, using the nibble management macro NBLBYT.

Non control characters cause the X coordinate location, stored in the open file variables, to be incremented but certain control characters are treated specially. A line feed character causes the current Y location to be incremented and a carriage return effects a zeroing of the current X location. A backspace character causes the X location to be decremented. Other control characters are simply stored in the segment but cause no change to be made to the current (X,Y) location.

The processing of vector data falls into two parts. Firstly, the vector data, in the form of (X,Y) coordinates, and a move/draw qualifier are extracted from the characters sent by the host. Secondly, this data is passed to the vector routine proper, which will optimise the storage required, whether short or long vector, knowing the current

(X,Y) location of the segment.

In order to minimise the amount of data sent to the GT4Ø, a compression scheme, which utilises the non-data part of the transmitted characters, is employed. This makes use of the possibility that only one of the coordinates may change or that the change in one may be small. The maximum value of an (X,Y) coordinate, which may be specified in long vector mode, occupies 1Ø bits of a 16 bit word. We can divide this into two parts, LO and HI, each requiring 5 bits, and name these parts of the (X,Y) coordinate pair xl,xh,yl,yh.

The choice of which of the options is selected is governed by bits 6 and 7 of the first character transmitted (bits 1-5 are data).

Ø1 - update xl,xh

1Ø - update xl,xh,yl

11 - update xl,xh,yh,yl

The values in the high order bits of the subsequent bytes and their meaning are as follows :-

Ø1 - a draw operation and this byte is the last byte.

1Ø - a move operation and this byte is the last byte.

11 - more bytes follow before last byte.

The sequences transmitted for particular commands are summarized below, with lower case characters indicating coordinate value bits.

Command	byte 1	byte 2	byte 3	byte 4
Move xl xh	ØØ1xxxxx	Ø1Øxxxxx		
Draw xl xh	ØØ1xxxxx	ØØ1xxxxx		
Move yl xl xh	Ø1Øyyyyy	Ø11xxxxx	Ø1Øxxxxx	
Draw yl xl xh	Ø1Øyyyyy	Ø11xxxxx	ØØ1xxxxx	
Move yl yh xl xh	Ø11yyyyy	Ø11yyyyy	Ø11xxxxx	Ø1Øxxxxx
Draw yl yh xl xh	Ø11yyyyy	Ø11yyyyy	Ø11xxxxx	ØØ1xxxxx

4.4 Light Pen and Tracking Cross Operation

In an interactive graphics system, one of the basic functions required is the ability of the user to identify an object displayed and to indicate an arbitrary position on the screen. The former would be used to select an item, or a particular part of an item, from amongst several displayed, whilst the latter might be used to position an item or specify the corners of an area which will be magnified or shown in greater detail. The identifying mode is described as 'picking' and the positioning mode is performed using an object known as a 'tracking cross'. In the GT4Ø both these functions are controlled by a 'light pen'.

The light pen consists of a narrow cylindrical tube with a light sensing phototransistor at one end. A lead terminating in a phono plug connects to the VR17 display where a module amplifies any signal produced. In normal use, by pointing the pen at the screen, the light producing such a signal will come from the phosphorescent image displayed.

If light pen interrupts are enabled, by setting a bit within a Set Graphic Mode instruction, then a light pen hit will cause the display processor to halt and interrupt the PDP-11. In the interrupt service routine two pieces of information are directly available. The display program counter (DPC), at location 172000, contains the address of the instruction following that on which the light pen hit occurred. The X and Y positions of the display at that time are obtained by reading locations 172004 and 172006. In addition, because of the way the display chain is implemented in GTX, also available at this point is the active display segment, whose existence was described in Chapter 2.

When a display segment is created, it is automatically made light pen sensitive, i.e. light pen interrupt enable is specified in the first graphic mode instruction. Thus all segments are always light pen sensitive. However, the IG/GT40 protocol defines when such light pen interaction details should be returned to the host system. The alternative to this would be to have all segments insensitive by default and to enable those which may be 'picked' by sending control characters to the GT40 for each. This would clearly be excessively verbose for a display with many pickable objects.

On reception of an ASCII DC2 <022> character, the command interpreter indicates to the user that a light pen pick operation may be performed. It does this by changing the cursor character in the type-ahead line to a solid square character (ASCII DEL <177>). Internally it sets a 'light pen want' indicator and a flag indicating that if a keyboard character is typed in the meantime, the light pen wait is to be aborted.

If a pick operation with the light pen is performed, the interrupt service routine first looks to see if a light pen hit is wanted. If it is, it further checks to see if a light pen hit has already been registered. If not, it sets the flag to 'light pen hit' and saves the (X,Y) hit coordinate and the active display file. In all other circumstances the hit is simply ignored. In the main processing loop, a check is made to see if a light pen hit has occurred. If so, a call to the CI is made with a null character.

If, however, the pick operation is aborted by the user typing a character, this character is simply passed to the CI. Since a null character is a control, the two return points within the CI will be different and allow the two possibilities to be distinguished. In the light pen case the display segment and the (X,Y) hit coordinate are sent to the host as a five character sequence

<seg#><high x><low x><high y><low y>

where the 10 bit coordinate data is broken down into two 5 bit items. In the abort case a single control character is transmitted.

In GTX the positioning function is performed with a tracking cross, which resembles a trapezium whose diagonals form a pair of cross wires. The design of the cross used is shown below.

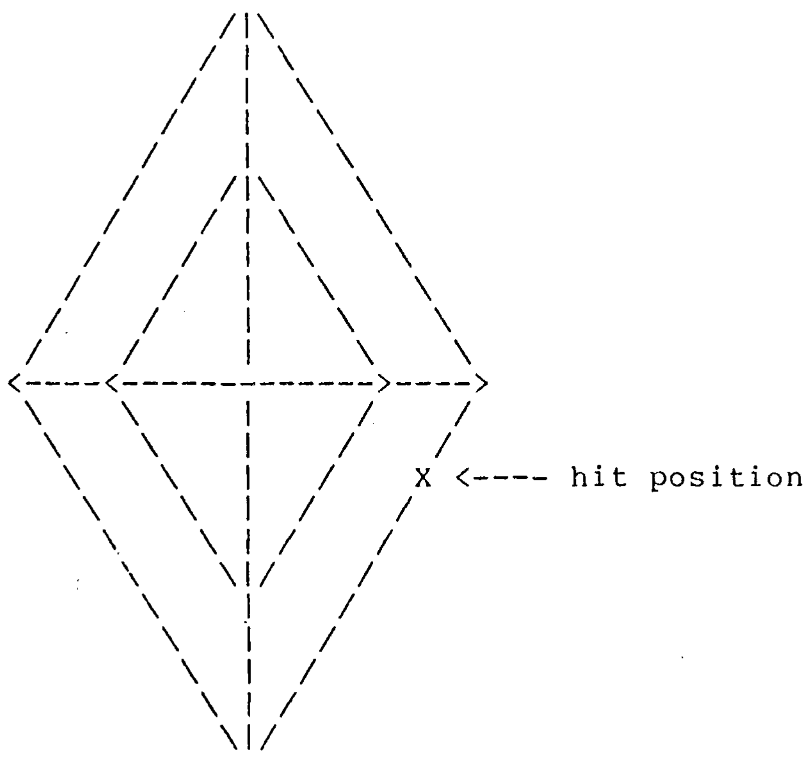

Fig 4.2

Tracking Cross

This cross is positioned with the light pen which 'drags' the cross, or more exactly the cross 'tracks' the light pen, by applying a series of small x and y increments to the cross centre.

All of the lines in the cross are sensitive to light pen interaction and a typical hit point is indicated at position (x,y). The hit will cause a processor interrupt through location 324 and as before the interrupt service routine reads the x and y hit coordinates from memory locations 172004 and 172006. As mentioned previously, an active display segment is maintained by the display interrupt routine. If this segment corresponds to the tracking cross, the following simple transformation is applied to the cross centre coordinates. Firstly the quadrant in which the hit occurred is found. Then the centre coordinates (X,Y) are changed by an amount (dx,dy), dependent on this quadrant. Typically, dx and dy are a few raster increments and define the fineness with which the tracking cross can be positioned.

Normally, the tracking cross is not linked into the display chain but on reception of an ASCII DC4 <Ø24> character, the tracking cross is inserted into the display chain and becomes visible on the screen. In addition a further modification is made to the flow of the display processor through the display segments. This causes the tracking cross to appear significantly brighter on the screen and enhances the tracking capability of the cross, which otherwise may track poorly when a lot of graphical data is displayed on the screen. What is done is to arrange that the display processor spends more time in the tracking segment compared to the other parts of the display file. In the display halt (DHALT) interrupt routine a check is made to see if the next segment to be displayed is the tracking cross. If so, then the DHALT address in the tracking segment is set to point to itself and a counter initialised to control the number of 'tight loops' made within the tracking segment. Subsequently whenever the next segment to be shown is the tracking segment, the counter is decremented until it reaches zero when the initial link is restored and the display processor continues with the other segments in the display file. The algorithm, expressed in a pseudo language, runs as follows :-

```
if next_segment == tracking_segment
   if dhalt_word_of_tracking_segment != tracking_segment
      save dhalt_word
      dhalt_word = tracking_segment
      initialise counter
   else
      decrement counter
      if zero
         restore dhalt_word
      endif
   endif
endif
```

In addition the cursor character in the type-ahead line is changed to a plus sign (+), chosen to resemble the cross wires of the tracking cross. Internally a flag is set indicating that any character typed will be made available immediately to the CI. This will occur when the user has satisfactorily positioned the cross and wishes its co-ordinates to be sent to the host. As before, the co-ordinates are sent to the host as a five character sequence, where the first character is the keyboard character typed by the user.

<keyc><high x><low x><high y><low y>

4.4 Utility Routines

Various commonly used utility subroutines were gathered together into a single module (control section) and made available to other modules, by declaring the routine names in a .GLOBL directive in the module containing common definitions.

These utility functions are :-

1) PRINTNO	print an internal value as an octal number on the console.
2) PRINTAS	print an ASCII string on the console. The string must be terminated by a null character.
3) I2D	convert an internal value to a decimal numeric string.
4) I2O	convert an internal value to an octal numeric string.

5) D2I

convert a decimal string to an internal value.

6) O2I

convert an octal string to an internal value.

PRINTNO calls I2O to convert the internal value to an octal string and then uses PRINTAS to output the string. D2I and O2I are called via a common name which decides which of them to use by looking at the first character of the string. If this is a zero then the number is assumed to be octal, if not decimal. Both octal and decimal conversion routines terminate when a value outside the allowable range is encountered.

CONCLUSIONS AND FUTURE DEVELOPMENT

5.1 General Conclusions

With the expansion of micro-processor based devices and the impact of what is loosely termed distributed computing, it is clear that the tools and techniques are ripe for the evolution of a series of intelligent terminals. To a large extent the development of this GT4Ø based system has been overtaken by the impact of cheaper and, in many cases, more sophisticated devices which are now becoming available. Clearly there can be no economic case made for the acquisition of this expensive equipment solely as an intelligent terminal system. However, given that the hardware was available, it was considered worthwhile to develop such a system.

No strong defence can be made of the use of assembler language in this project, apart from easy availability. A possible candidate, which is available under the UNIX system, is the programming language MODULA (ref 12,13). A system could be written in MODULA, compiled and linked under UNIX, and down line loaded into the PDP 11/1Ø. Primarily designed as a multi-tasking language for process control, MODULA provides many high level data and programming structures, features very suited to this application. However, Wirth, the author of MODULA, describes the difficulty of handling the GT4Ø in one of his case studies (ref 14). Other work, using the language, has controlled the GT40 with a certain amount of success (ref 15).

In any substantial programming project the elements of fault finding, diagnostic testing and error analysis, collectively known as 'debugging', should not be overlooked. In many ways this is a more difficult exercise than the project itself. At several points in the development, the need for some form of debugging sub-system seemed to arise. In practice a combination of diagnostic print and information logging in memory, for later post-mortem examination, was used in an ad-hoc fashion for each non-trivial error encountered. This might be considered a symptom of the need for a high level language implementation, where debugging aids and diagnostic information are more directly available. However, several of the high level languages considered suitable for system programming have quite large run time requirements, for example PASCAL (ref 16). MODULA on the other hand, has a minimal run time support, but provides no debugging facilities. It is certainly true that one of the most difficult errors to isolate in GTX, which required a specific combination of factors within the system in order to appear, could not easily have been located by a facility such as breakpoint debugging.

One question which begs itself is 'just how useful is a dual session terminal?'. It is true that for most users the two host capability was a luxury and use was mainly made of the extra facilities to a single host. However, for systems programming and communications work, the dual connection was of great benefit. One such example was in the midst of the development of a file transfer facility between UNIX and MTS. At that stage both of these systems maintained detailed logs of information about the transfers and, when a tranfer had failed, it was necessary to look at both logs to determine the cause of failure. In this circumstance the two session feature was most useful and allowed the two logs to be compared adjacent to one another. Indeed had this capability been available earlier in that project, it may have led to a more interactive testing facility, with commands to both ends of the transfer link originating from the 'same' terminal.

Testing of the basic functioning of the system as a two 'session' terminal was performed as it was developed, by connecting it to two host systems. Use was sometimes made of the 'local' mode, to echo characters locally, when neither of the hosts was available or when reproducibility was important. The graphics side of the system was exercised by the graphics package *IG in MTS, using the example programs in IG:EXAMPLES, or by test programs running in UNIX, which together tested the vector generation, light pen and tracking cross functions.

5.2 Future Development

One of the greatest shortcomings of attempting to use standard equipment, as supplied with the system, was the lack of **programmed function keys** on the terminal keyboard of the LA36 DECwriter. It would be feasible, if somewhat irksome, to arrange for the use of more control characters to perform the equivalent functions. The disadvantage is that two key depressions are required and difficulty is encountered in remembering the keys involved, since special markings are not available. With a certain amount of rewiring, it may be possible to arrange for the usually unused numeric keypad on the DECwriter to generate control codes.

The use to which such function keys might be put are varied. One use might be to enable character strings to be defined and subsequently be recalled into the type-ahead window with a single keystroke. Thus often used commands may be sent with a minimum of typing. Another use is to provide vertical scrolling through the 'session', so that earlier text could be 'replayed' on the screen.

Additional cursor control could be provided to allow the user to 'mark' a line on the screen and have it appear in the type-ahead window. This 'line re-entry' would be useful for resubmitting a previous command.

In the current system, the 2 RKØ5 disk units, of 2.5Mb capacity each, are unused. Clearly they provide ample secondary storage for several possible uses. One of these might be to hold the output sent to the system, so that earlier 'pages' in the sessions could be recalled and reviewed. In combination with a line re-entry facility, this would be a useful addition. A second more obvious use is simply to use the disk as a filestore, and provide a means for redirecting the host output and input, to or from one of these local files. This would provide a simple means of file transfer and would allow, for example, local editing of remote files to be performed in visual mode under RT-11.

To a large extent the system evolved, rather than developed toward a specified end product, which is a mixed blessing in some ways. Certainly the evolution was sufficiently flexible to respond to users recommendations and experiences, which made for a sense of 'usefulness' in the project. Equally, the possible enhancements which have just been mentioned, may impress a feeling of a project not yet fulfilled.

APPENDIX

A.1 Structured Macros

```
.macro   push   arg1,arg2,arg3,arg4,arg5,arg6,err
.narg   zzzcnt
.iif   gt zzzcnt-6      .error   zzzcnt; too many cells pushed
.iif   ge zzzcnt-1      mov      arg1, -(sp)
.iif   ge zzzcnt-2      mov      arg2, -(sp)
.iif   ge zzzcnt-3      mov      arg3, -(sp)
.iif   ge zzzcnt-4      mov      arg4, -(sp)
.iif   ge zzzcnt-5      mov      arg5, -(sp)
.iif   ge zzzcnt-6      mov      arg6, -(sp)
.endm   push

.macro   pop   arg1,arg2,arg3,arg4,arg5,arg6,err
.narg   zzzcnt
.iif   gt zzzcnt-6      .error   zzzcnt; too many cells popped
.iif   ge zzzcnt-1      mov      (sp)+, arg1
.iif   ge zzzcnt-2      mov      (sp)+, arg2
.iif   ge zzzcnt-3      mov      (sp)+, arg3
.iif   ge zzzcnt-4      mov      (sp)+, arg4
.iif   ge zzzcnt-5      mov      (sp)+, arg5
.iif   ge zzzcnt-6      mov      (sp)+, arg6
.endm   pop

.macro   lose   n
.iif   le n    .error   n; arg for 'lose' < 1
.if   eq n-1
```

```
  tst (sp)+            ; drop one item from stack
.mexit
.endc
.if  eq n-2
  cmp (sp)+, (sp)+     ; drop two items from the stack
.mexit
.endc
  add #n+n, sp
.endm  lose


.macro  call  subr
  jsr pc,subr
.endm  call


.macro  return
  rts pc
.endm  return


.macro  blkon
.mcall  loop,break,repeat,if,else,endif,blkoff
.mcall  $setlp,$getlp,$setbn,$getbn,$conbr,$genlb
zzzzzn = Ø  ; this is the nest level (if-endif and loop-repeat)
zzzlbn = Ø  ; label sequence number for if ... endif
zzzlpn = Ø  ; label sequence number for loop ... repeat
.endm  blkon


.macro  blkoff
.iif  ne,zzzzzn        .error ; unclosed block in program
```

```
.endm  blkoff


.macro  loop
.iif  ndf,zzzzzn       .error ; no blkon preceding loop
zzzzzn = zzzzzn+1
zzzzzl = zzzzzn  ; for 'break' nested inside if ... endif
zzzlpn = zzzlpn+1
  $setlp      zzzlpn,%zzzzzl
  $genlb      qx,%zzzlpn
.endm  loop


.macro  break cond
.iif  ndf,zzzzzn       .error ; no blkon preceding break
.iif  le,zzzlpn        .error ; break illegal outside loop
  $getlp      rln,%zzzzzl
  $conbr      cond,qy,%rln,x
.endm  break


.macro  repeat         cond
.iif  ndf,zzzzzn       .error ; no blkon preceding repeat
.iif  le,zzzlpn        .error ; repeat has no preceding loop
  $getlp      rln,%zzzzzn
  $conbr      cond,qx,%rln,x
  $genlb      qy,%rln
  $setlp      Ø,%zzzzzn
zzzzzn = zzzzzn-1
zzzzzl = zzzzzl-1
.endm  repeat
```

APPENDIX

```
.macro  $setlp        lpn,bn
zzqx'bn = lpn
.endm  $setlp


.macro  $getlp        lpn,bn
lpn = zzqx'bn
.endm  $getlp


.macro  if    cond
.iif  ndf,zzzzzn       .error ; no blkon preceding if
zzzzzn = zzzzzn+1
zzzlbn = zzzlbn+1
  $setbn      zzzlbn,%zzzzzn
  $conbr      cond,qq,%zzzlbn
.endm  if


.macro  else
.iif  ndf,zzzzzn       .error ; no blkon preceding else
  $getbn      rbn,%zzzzzn
.iif  lt,rbn  .error ; else follows else with no endif or if
.iif  eq,rbn  .error ; out of sync
.if  gt,rbn
  $conbr      ,qz,%rbn
  $genlb      qq,%rbn
  $setbn      <-rbn>,%zzzzzn
.endc
.endm  else
```

APPENDIX

```
.macro  endif
.iif  ndf,zzzzzn      .error ; no blkon preceding endif
.iif  le,zzzzzn       .error ; endif unmatched by if
  $getbn      rbn,%zzzzzn
.iif  gt,rbn  $genlb  qq,%rbn
.iif  lt,rbn  $genlb  qz,%<-rbn>
.iif  eq,rbn  .error ; out of sync
  $setbn      Ø,%zzzzzn
zzzzzn = zzzzzn-1
.endm  endif


.macro  $setbn        lbn,bn
zzqq'bn = lbn
.endm  $setbn


.macro  $getbn        lbn,bn
lbn = zzqq'bn
.endm  $getbn


.macro  $conbr        cond,pref,bn,x
.if  b,<cond>
  br  pref'bn
.iff
.if  nb,<x>
  b'cond      pref'bn
.iff
.iif  idn,<cond>,<eq> bne     pref'bn
```

```
.iif   idn,<cond>,<ne> beq      pref'bn
.iif   idn,<cond>,<lt> bge      pref'bn
.iif   idn,<cond>,<ge> blt      pref'bn
.iif   idn,<cond>,<gt> ble      pref'bn
.iif   idn,<cond>,<le> bgt      pref'bn
.iif   idn,<cond>,<pl> bmi      pref'bn
.iif   idn,<cond>,<mi> bpl      pref'bn
.iif   idn,<cond>,<cs> bcc      pref'bn
.iif   idn,<cond>,<cc> bcs      pref'bn
.iif   idn,<cond>,<hi> blos     pref'bn
.iif   idn,<cond>,<lo> bhis     pref'bn
.iif   idn,<cond>,<his> blo     pref'bn
.iif   idn,<cond>,<los> bhi     pref'bn
.endc
.endc
.endm  $conbr


.macro  $genlb        pref,rbn
pref'rbn:
.endm  $genlb

.macro  freeze        pri
.narg  argcnt
mov    @#psw,-(sp)
.iif eq,argcnt  mov   #340,@#psw
.iif eq,argcnt-1  mov pri,@#psw
.endm freeze


.macro  thaw
```

```
  mov -(sp),@#psw
.endm thaw
```

A.2 BNF Definition of IG/GT40 Protocol

```
<file>   :=  <head><data>

<head>   :=  <soh><seg#><data>

<seg#>   :=  <char>

<data>   :=  <stext>[<vec><text>] | <stext>[<vec><text>]<vec>

<stext>  :=  [<char>]

<vec>    :=  <so>[<char>]

<text>   :=  <si>[<char>]

<char>   :=  <040>|<041>|<042> ...... <175>|<176>|<177>

<so>     :=  <016>

<si>     :=  <017>

<soh>    :=  <001>

<stx>    :=  <002>
```

Square brackets [] enclose elements which can be repeated an arbitrary number of times, including zero.

A.3 System Generation

Generation of the GTX software can be obtained by typing the following commands to the RT-11 operating system :-

```
R MACRO
*GTMAIN = GTMAC,GTMAIN
*GTDEV  = GTMAC,GTDEV
*GTSCRN = GTMAC,GTSCRN
*GTGRAF = GTMAC,GTGRAF
*GTCI   = GTMAC,GTCI
*GTUTIL = GTMAC,GTUTIL
^C

R LINK
GTX,GTX = GTMAIN,GTDEV,GTSCRN,GTGRAF,GTCI,GTUTIL
^C
```

The resulting system can then be run by the command

```
R GTX
```

REFERENCES

1) IBM 327Ø Display Station, MTS Volume 4, "Terminals and Tapes" , University of Michigan.
"IBM 327Ø Information Display System Component", form GA27-2749.

2) "The UNIX Time-Sharing System", D.M.Ritchie & K.Thompson, CACM, Vol 17 no. 7, (1974) pp. 365-375

3) "The Michigan Terminal System", D.W.Boettner & M.T.Alexander, Proc. of IEEE, Vol 63, No. 6, (1975) pp. 912-918

4) "Syntactic definition and parsing of molecular formulae, Part 2: Graphical synthesis of molecular formulae for data base queries", P.G.Barker & P.S.Jones, Computer Journal, Vol 21 no. 3, (1978) pp. 224-232

REFERENCES

5) "Use of *IG in MTS", NUMAC (April 1977). Michigan Computing Centre Memo no. 299

6) "The Design and Implementation of INGRES", ACM Trans. on Database Systems 1,3 (Sep 1976), pp. 189-222

7) "PDP11 Processor Handbook 1978-79", Models Ø4/Ø5/1Ø/35/4Ø/45, Digital Equipment Corporation

8) "GT4Ø/42 Users Guide", Digital Equipment Corporation, EK-GT4Ø-OP-ØØ2

9) "VT11 Graphic Display Processor", Digital Equipment Corporation, EK-VT11-TM-ØØ1

1Ø) "Terminals & Communications Handbook 1978", Digital Equipment Corporation

REFERENCES

11) "RT-11 Software Support Manual", Digital Equipment Corporation, DEC-11-ORPGA-B-D

12) "Modula : a Language for Modular Programming", N.Wirth, Software-Practice and Experience, Vol 7 (1977), pp. 3-35

13) "Design and Implementation of Modula", N.Wirth, Software-Practice and Experience, Vol 7 (1977), pp. 67-84

14) "The Use of Modula", N.Wirth, Software-Practice and Experience, Vol 7 (1977), pp. 37-65

15) "An assessment of Modula", J.Holden and I.C.Wand, Report YCS.16 (1978), Department of Computer Science, University of York

16) "The programming language Pascal", N.Wirth, Acta Informatica, Vol 1, no. 1, (1971) pp. 35-63

REFERENCES

17)

"The Programming Language Concurrent Pascal", P. Brinch Hansen, IEEE Trans. on Software Engineering, Vol SE-1, no. 2, (1975) pp. 199-207