SYSTEM IMPLEMENTATION NOTES FOR DOS-BASED PC-McIDAS

Jonathan R. Ide

Space Science and Engineering Center
University of Wisconsin - Madison

November, 1987

Space Stone
UWSSEC Publication
No. 87.11.I1

TABLE OF CONTENTS

TABLE OF CONTENTS

PREFACE

The purpose of these notes is to describe the most important elements of the **systems** programs underlying the implementation of PC-McIDAS workstations under DOS 3.X. **Applications** programs are not discussed except in the most general way, nor are the higher-level Fortran utilities that are used for such things as accessing PC-McIDAS data bases. Lower-level utilities are discussed in some detail. The emphasis is on the underpinnings of the workstation implementation **per se**.

Even within the intended scope of these notes, a complete, detailed description is not possible. I have tried at least to orient the reader sufficiently to allow him/her to see the big picture and get some idea of where in the source code to look to answer specific questions or problems. I have tried also to pay particular attention to the most impenetrable aspects of the PC-McIDAS implementation, giving them a more detailed treatment.

These notes have been written in some haste. I have tried to be accurate, but the final authority must be the source code itself. Also, there are certain topics I wanted to cover but time does not permit. These include: BIOS function interface entry points; command line editting and the command line stack; the drop-down menu HELP interface; interactions with the soft tablet interface software; interactions with voice interface software; the scheduler; and the menu system and function keys. However, I believe I have covered the topics that will be of greatest benefit to future maintainers of the software.

For a more general treatment of some of the issues described herein, see the following papers:

PC-Based McIDAS. Jonathan Ide. Preprint Volume, 3rd International Conf. for Meteorology, Oceanography, and Hydrology, New Orleans, LA, AMS.

The UNIDATA PC-McIDAS Workstation -- A Technical Discussion. Jonathan Ide. Preprint Volume, 4th International Conf. for Meteorology, Oceanography, and Hydrology, Anaheim, CA, AMS.

Man-Machine Interfaces Developed for a PC-Based McIDAS Workstation. Jonathan Ide, Russell Dengel, and Robert Krauss. Preprint Volume, 3rd International Conf. for Meteorology, Oceanography, and Hydrology, New Orleans, LA, AMS.

Flexible Menu Creation for the Unidata PC-McIDAS Workstation. Jonathan Ide. Preprint Volume, 4th International Conf. for Meteorology, Oceanography, and Hydrology, Anaheim, CA, AMS.

The University of Wisconsin-Madison UNIDATA Broadcast Experiment. Russell Dengel, Jonathan Ide, Raymond Lord, David Santek, Thomas Whittaker, and J. T. Young. Preprint Volume, 4th International Conf. for Meteorology, Oceanography, and Hydrology, Anaheim, CA, AMS.

Data Management on the UNIDATA PC-Based Workstation. Gail Dengel, Russell Dengel, Jonathan Ide, and Raymond Lord. Preprint Volume, 4th International Conf. for Meteorology, Oceanography, and Hydrology, Anaheim, CA, AMS.

PC-McIDAS DESIGN GOALS

Some PC-McIDAS Design Goals

0. Provide a superset of the functionality of existing (non PC-based) McIDAS terminals.

1. Provide a systems environment in which mainframe McIDAS commands can be ported to the workstation with minimum modifications to mainframe source code.

2. Isolate hardware dependencies, and support a variety of hardware configurations. Allow the pertinent hardware components (graphics devices, communications hardware, pointing devices, etc.) to be permuted with minimum impact on applications software. Enable new devices to be incorporated easily in the future, again with minimum impact on applications software. Allow a given workstation's configuration to be changed easily.

In other words, provide for the development of an entire **family** of related workstations that all use the same scanner, applications programs, and user interfaces, and allow hardware drivers to be interchanged flexibly.

3. Isolate operating system dependencies. Attempt to minimize the impact on **applications** software of the inevitable future conversion to other operating systems. (Naturally, the impact on **systems** software will be more drastic.)

4. Develop new, highly interactive user interfaces. Provide a framework in which new interfaces can be easily incorporated into the system in the future.

How the Goals Have Been Met

1. There is a so-called "utility layer" used by mainframe applications software -- LW file routines (LWI, LWO, etc.), routines for fetching parameters (IPP, CKWP, etc.), etc. To the fullest extent possible, this utility layer was reproduced in PC-McIDAS. Calling sequences were kept intact wherever possible, even though the routines themselves had to be rewritten. Presenting the familiar utility layer entry points to applications programs made it possible for mainframe applications to "feel at home" in PC-McIDAS with minimum modifications to source code.

MicroSoft Fortran currently does not support the full Fortran 77 standard. The greatest deficiencies are in the area of string manipulations. A set of utilities were created to help applications programmers perform the various kinds of functions supported by mainframe Fortran but not by Fortran on the AT.

2. Applications communicate with hardware device drivers through a virtual interface known as SYSCOM (System Common Area), analogous to UC (User Common) on the mainframe. To change the current frame number, for example, an application changes the appropriate value in SYSCOM. The application need not concern itself with how or on what kind of device the new frame actually gets displayed.

In addition, a limited number of entry points are available to applications to perform such functions as drawing a pixel on a graphics frame. However, these entry points are implemented in such a way that no hardware dependent code is actually linked in with the application. Instead, the entry point activates a software interrupt. The interrupt handler appropriate to the particular hardware device in use is installed separately.

All hardware dependent drivers are isolated in resident interrupt handlers. These drivers are self-contained modules that are installed at boot-time. Setting a workstation up for a particular hardware configuration amounts to providing a mechanism to ensure that the appropriate versions of the drivers get installed.

A configuration program (CONFIG) is provided which steps the user through a series of questions that define the workstation's configuration. CONFIG automatically sets up the batch files needed to configure the workstation according to the user's specifications. The same software installation diskettes are used for all PC-McIDAS workstations.

3. DOS function calls are hidden from applications programs. PC-McIDAS provides a set of Fortran-callable entry points that perform DOS functions. For many applications, porting to another operating system will require only that these DOS function subroutines be recast in the new environment.

4. Various user interfaces have been developed. They interact with the PC-McIDAS scanner via SYSCOM, so old user interfaces can be dropped from the system or new interfaces developed with minimal impact on the system as a whole.

SUMMARY OF INTERRUPT VECTOR USAGE

The following interrupt vectors are modified by PC-McIDAS:

INT 10H -- Replacement for the BIOS video interrupt. The BIOS code is re-vectored to INT 62H; VIDEO.EXE is installed under INT 10H. VIDEO provides INT 10H functionality under the text window interface. Certain functions not handled by VIDEO are passed through to the BIOS code via INT 62H. (See VIDEO.ASM.)

INT 16H -- KBIOSF, the replacement for the BIOS keyboard interrupt. The interrupt vectors for both KBIOSF and the BIOS keyboard handler are kept in SYSCOM. KBIOSF is installed during PC-McIDAS run-time initialization, de-installed when PC-McIDAS exits. (See KBIOSF.ASM.)

INT 1CH -- The "tick" for TVCTRL in EGA/VGA-based workstations. INT 1CH is the "time-of-day" interrupt, triggered 18.2 times per second. (See TVEGA.ASM.)

INT 60H -- SYSCOM interface. Access to SYSCOM is provided by the various functions of INT 60H. (See SYSCOM.ASM.)

INT 61H -- Low-level communications drivers. In ProNet-based workstations, the various functions of INT 61H provide access to the ProNet interface hardware. (The driver is PNETINT.EXE; see PINTLNK.ASM, COMINT.ASM and PNETINT.C) In async-based workstations, INT 61H provides access to the serial port. (The driver is ASYNC1.EXE or ASYNC2.EXE, depending on whether serial port 1 or 2 is used; see ASYNC1.ASM and ASYNC2.ASM, respectively.) High-level communications functionality (e.g. decoding and processing packets, formulating reply messages) are handled by the drivers installed under INT 64H.

INT 62H -- BIOS video interrupt. Re-vectored from INT 10H (q.v.).

INT 63H -- Graphics/imagery drivers. Functions to define a graphics window, draw a graphics point or line segment, or load an image line. (For EGA/VGA-based workstations, the driver is PVEGA.EXE; see PVEGA.ASM. For tower-based workstations, the driver is PVSSEC.EXE; see PVSSEC.ASM.)

INT 64H -- High-level communications drivers. These drivers process entire packets, leaving it to the low-level drivers to interface to the comm hardware. Called from TVCTRL. (For ProNet-based workstations, the driver is COMMP.EXE; see COMMP.FOR, PCOMM.ASM, PUTPCK.ASM, and GETPCK.ASM. For async-based workstations, the driver is COMMA.EXE; see COMMA.FOR, ACOMM.ASM, PUTPCK.ASM, and GETPCK.ASM. For stand-alone workstations, the driver is COMMN.EXE; see NCOMM.ASM.)

INT 65H -- Software interface to TVCTRL. See chapter on TVCTRL for description of functions. (For EGA/VGA-based workstations, see TVEGA.ASM. For tower-based workstations, see TVSSEC.ASM.)

INT 66H -- Driver for text window interface. Refreshes text window; handles PageUp, PageDown, etc.; handles toggle among text windows, soft tablet, and/or EGA/VGA frames; etc. Called from TVCTRL. (Driver is SCRINI.EXE; see SCRINI.FOR, SCREENS.ASM, and CLRPAG.ASM.)

INT 67H -- The use of this interrupt is up in the air at the time of this writing. It may be made available to the voice-recognition handler, which would be called from TVCTRL. However, INT 67H is used by the Lotus-Intel-Microsoft Expanded Memory Management system, so it may be advisable to leave it alone and use another mechanism to activate the voice-recognition handler.

INT 77H -- The "tick" for TVCTRL in tower-based workstations. Triggered by IRQ15, which in turn is triggered by the TV timing interrupt in the tower. (See TVSSEC.ASM.)

A FEW HARDWARE CONSIDERATIONS

PC-McIDAS applications programs are isolated from hardware specifics, for the most part. The systems code, however, needs to interact directly with the hardware in various ways. Hardware specifics will be discussed in some detail in the chapters on the various device drivers. The purpose of this chapter is to give a brief overview of certain hardware considerations that have impacted the PC-McIDAS system design in significant ways.

Interacting with the "Tower"

PC-McIDAS is designed to operate in a variety of hardware configurations. One such configuration couples the workstation computer (IBM AT) with an SSEC/Dataram video/graphics display unit, otherwise known as a "tower".

The tower was the heart of the "old" (pre-PC) McIDAS terminal. It contains an 8085 microprocessor that executes code stored in ROM. The 8085 was used to control the display in various ways: looping control, cursor drawing, joystick monitoring, color enhancements, etc. In the pre-PC McIDAS terminal, communications routines resided in the tower and comm packets generated on the host were handled directly by the tower firmware.

In a PC-McIDAS workstation, it is necessary for the PC-McIDAS system to intercept all comm packets coming from the host. PC-McIDAS needs to know what is going on and needs to be able to communicate with the host directly. Naturally, then, the PC-McIDAS communications software runs in the AT, no longer in the 8085.

Similarly, PC-McIDAS needs to be able to control many of the functions formerly handled by the 8085: looping, cursor positioning, etc.

Two possible design paths were considered. It would have been possible to do away completely with the 8085 and its associated firmware and control the display completely from the AT. Alternatively, the 8085 and its firmware could be retained, and the AT could initiate actions in the tower by passing to the tower the same kind of comm packets the tower firmware was used to receiving. After much discussion, the latter approach was taken. It was felt that reproducing the firmware functionality on the AT was a non-trivial task with relatively little to recommend it beyond increased ease of maintenance.

The hardware interface between the tower (which uses a MultiBus architecture) and the AT (which uses the AT bus architecture) is through a pair of interface cards manufactured by Bit3 Corp. One card resides in the AT, the other in the tower. The interface is via a dual-ported memory that resides at segment address 0D000H in the AT.

Two buzzwords are often heard in connection with the AT-tower interface. The first is "Bit3 card". This refers to the interface cards mentioned above. Bit3 is a brandname only; there is no other significance to the name. The second buzzword is "02/03 protocol". This refers to the comm protocol that defines the packets passed between the AT and the tower (and between the host and old-style McIDAS terminals). The name stems from the fact that each packet begins with an 02 byte and ends with an 03. For communications between the host and a PC-McIDAS workstation, a new protocol has been instituted. It is referred to as the "F0 protocol". See the chapter on communications.

One significant fact about the Bit3 card: at the time of this writing there does not exist a version interfacing the MultiBus to the MicroChannel Architecture, so it is not yet possible to couple a tower to a PS/2.

Communications Hardware

PC-McIDAS supports two principal communications modalities:

- ProNet LAN
- Asynchronous serial connection.

Async comm can be via telephone, direct line, or satellite broadcast.

ProNet workstations require a ProNet interface card in the AT. Async workstations use the standard IBM serial/parallel adapter card or the serial port on the AST Advantage card or other extended memory card.

At the time of this writing no ProNet interface card exists that can be used in the PS/2.

Extended Memory

DOS 3.x does not allow programs to run in memory above 640K. BIOS, however, does provide a means of moving data to or from memory above 640K. PC-McIDAS systems tasks make extensive use of the ability to store data in so-called "extended memory" -- memory above 1 megabyte.

There is a catch, however. In order to access extended memory, the BIOS routine must switch the 80286 into protected mode, do the move, and then reset the 80286 to get it back into real mode. This last step involves the keyboard controller, believe it or not, and is quite slow. Interrupts are disabled throughout, since any interrupt handlers present will have been written to run in real mode. The result is that if high-speed interrupt processing is also going on, interrupts will be lost when extended memory is accessed. The impact of this was felt in connection with async comm handling. The problem was solved by using XON/XOFF control -- see the chapter on communications.

SYSCOM -- SYSTEM COMMON AREA

PC-McIDAS processes communicate with one another via a resident "mailbox" known as SYSCOM (System Common Area). Various fields in SYSCOM are defined to have particular meanings. For example, there is a particular byte allocated for storing the current image frame number. PC-McIDAS processes that need to read or modify the image frame number do so by accessing this byte in SYSCOM.

Why is SYSCOM Needed?

The SYSCOM mechanism is important for two reasons.

First, it provides a means for interprocess communication, something that is not supported by single-tasking operating systems like DOS. Not only are global data values stored there, but SYSCOM is also used for storing various flags and semaphores used to synchronize tasks and communicate between processes.

Second, SYSCOM makes it possible to isolate hardware dependent code from applications. An application can modify the current image frame number, for example, without having to know how or on what kind of device the image frame will actually be displayed.

SYSCOM Structure

SYSCOM is a fluid construct, constantly being extended and modified. It is highly desirable that it be structured in a way that lends itself to frequent extensions and modifications and still allow some internal consistency and rationality in the way SYSCOM contents are laid out.

For this reason, SYSCOM is structured as a sequence of blocks of various sizes. Contents are addressed by their block number and offset within block. This means that any given block can be enlarged (or shrunk) without causing a change in the address of any existing SYSCOM item.

Moreover, each block can be defined to contain items that have a common purpose or meaning. This brings some consistency and order to the SYSCOM layout. At the time of this writing, the following SYSCOM blocks have been defined:

Block 0: Terminal Control Block. State of the workstation. Number of frames, current frame number, cursor size and color, type of display hardware being used, screen size, etc. Also various flags and semaphores basic to workstation operation.

Block 1: Looping Control Block. Image loop, graphics loop, and opposite loop definitions, and dwell rates. Supports "random" looping of frames.

Block 2: Applications Data Interchange Block. Values defined ad hoc for interprocess communications (IPC) among applications. (Block 0 handles IPC for systems processes fundamental to control of the workstation itself.)

Block 3: Command Parameter-Passing Block. Used by the scanner to pass parameters to a PC-McIDAS command.

Block 4: User Interface Block. Values used by the system to control user interfaces and by user interfaces to pass commands back to the system.

Block 5: Voice Interface Block. Similar to Block 4, but specific to voice recognition interfaces.

Block 6: Command Stack Block. Used to store the last 10 commands entered in the current PC-McIDAS session.

Block 7: Frame Palette Block. Used to store color palettes for frames in EGA/VGA implementations.

Block 8: Communications File Pool Block. Used to maintain a pool of temporary files used by communications software.

The definitions of the contents of the various SYSCOM blocks are detailed in the Appendix.

SYSCOM Implementation

The data storage area comprising SYSCOM needs to be accessible by all system tasks and applications programs. Moreover, some means must be provided by which these processes can read or write values in SYSCOM. Ideally, neither the data storage area nor the means for accessing it should be linked into any system task or applications program. One would like to be able to modify the structure of SYSCOM (enlarging a block, for example) without having to relink anything else.

To meet the above requirements, SYSCOM was implemented in the following way. A resident interrupt handler (SYSCOM.EXE -- source code in SYSCOM.ASM) was created, providing the means for accessing SYSCOM. The data storage area itself is the local data segment of SYSCOM.EXE. SYSCOM.EXE was installed under user interrupt vector 60H. To modify SYSCOM, then, all one needs to do is modify SYSCOM.EXE and install the modified version. No applications or other system tasks are affected.

Data is laid out within the local data segment as follows. First there is a block which is not included in the block numbering scheme detailed above. I.e. it precedes block 0. It consists of segment address pointers to the other SYSCOM blocks. It is followed by the other SYSCOM blocks, in order. Because the various blocks are accessed via segment address pointers and are stored contiguously, the length of each block (including the block of pointers) must be a multiple of 16.

Accessing SYSCOM

Assembler routines access SYSCOM via INT 60H. The particular function performed by INT 60H is determined by various register values, as follows:

Register AL -- Function code.
- 0 = read
- 1 = write
- 2 = get segment address of SYSCOM block
- 3 = initialize (zero out the data area)

Register AH -- SYSCOM block number (0-based).

Register BX -- Offset (bytes, 0-based) within block.

Register CX -- Length of item (in bytes).

Registers DS:DI -- Pointer to the address at which the return value is to be stored (if AL = 0 or 2) or at which is stored the value to be written (if AL = 1).

Note: If AL = 2 and AH = 0FFH, the value returned is the segment address of the SYSCOM pointers block.

Various Fortran-callable entry points are provided to enable applications programs to access SYSCOM. These access routines are implemented as assembler modules that set up the registers appropriately and activate INT 60H. The calling sequences are as follows:

CALL **POKEB**(BLOCK,OFFSET,VALUE) -- Write a byte value

IVAL=**LOOKB**(BLOCK,OFFSET) -- Read a byte value

CALL **POKEW**(BLOCK,OFFSET,VALUE) -- Write a 2-byte word

IVAL=**LOOKW**(BLOCK,OFFSET) -- Read a 2-byte word

CALL **POKEDW**(BLOCK,OFFSET,VALUE) -- Write a 4-byte word

IVAL=**LOOKDW**(BLOCK,OFFSET) -- Read a 4-byte word

CALL **POKES**(BLOCK,OFFSET,SOURCE,SOFFST,LENGTH) -- Write a string of bytes, located at offset SOFFST within the byte array SOURCE

CALL **LOOKS**(BLOCK,OFFSET,DEST,DOFFST,LENGTH) -- Read a string of bytes, returning them to offset DOFFST within the byte array DEST.

IVAL=**LOOKSB**(BLOCK,OFFSET) -- Return a signed byte value (i.e. sign-extend the result)

IVAL=**LOOKSW**(BLOCK,OFFSET) -- Return a signed 2-byte value (i.e. sign-extend the result).

The VALUE and IVAL variables are assumed to be declared as 4-byte integers (the standard for all PC-McIDAS integers).

Device drivers generally need to access SYSCOM quite a lot. To improve performance, they do not call INT 60H for each access. Instead, when they are installed they call INT 60H with AH=2 to obtain the segment addresses of the SYSCOM blocks they will need to access. From then on, they access SYSCOM locations directly by applying the segment address and the known offset of the item being accessed.

Modifying SYSCOM

To modify the definition of a SYSCOM block -- e.g. to assign a definition to a heretofore unused byte -- you need only note the new definition in the file SYSCOM.DEF that contains the SYSCOM layout. Management of SYSCOM.DEF is not a software issue, but an issue of maintaining consistency among the various programmers who use SYSCOM. There needs to be a single, canonical copy of SYSCOM.DEF.

From time to time, however, it is necessary to expand a SYSCOM block or add a new block. To expand an existing block, you need only change the appropriate constant definition at the beginning of SYSCOM.ASM, then reassemble and relink. To add a block, add a new constant definition and a new block in the data segment definition and add code under the SYSSETUP label at the end of SYSCOM.ASM. Use the existing code as a guide. Be sure that all block lengths are multiples of 16, and if you are adding a block be sure that the pointers block is still big enough to hold all the pointers.

TV CONTROL

Hardware-specific drivers for controlling the display hardware and associated peripherals (e.g. mouse, joysticks) are given the generic designation TVCTRL ("TV control"). Individual TVCTRL drivers are created for each display device supported. Two such drivers exist at the time of this writing:

TVSSEC - controls SSEC/Dataram "tower"

TVEGA - controls IBM EGA (Enhanced Graphics Adapter) or IBM VGA (Video Graphics Array)

Each is a resident interrupt handler written in assembly language.

General Principles of TV Control

Certain considerations apply to all versions of TVCTRL; they will be discussed in this section. Specifics of the individual TVCTRL implementations will be discussed in the following sections.

TVCTRL Functions

TVCTRL is driven by a periodic interrupt, or "tick" (18.2 hertz for TVEGA, 30 hertz for TVSSEC). On each tick, TVCTRL inspects the relevant values in SYSCOM and changes the workstation state as needed to reflect the current SYSCOM entries. For example, if the image frame number has changed since the last tick, TVCTRL causes the new image frame to be displayed on the screen.

TVCTRL governs the pointing devices. If a mouse is present, TVCTRL polls INT 33H to monitor mouse movement and mouse button presses. Mouse status is kept up-to-date in SYSCOM. (Joysticks apply only to the TVSSEC implementation of TVCTRL so will be discussed below.)

TVCTRL manages image and graphics looping, using the loop definitions and dwell rates stored in the Looping Control Block in SYSCOM. It modifies the current image and graphics frame numbers in SYSCOM as the loop proceeds.

TVCTRL handles other functions related to the image/graphics display, such as positioning and drawing the cursor.

Finally, TVCTRL handles various single-letter commands: A, B, J, K, L, M, O, P, V, W, Y, Z. TVCTRL monitors the keyboard input through the keyboard filter mechanism (see the chapter entitled "The Keyboard Filter") so it can give immediate action to single-letter commands entered via the Alt key. Each implementation of TVCTRL includes a jump table that governs the handling of the various single-letter commands.

Various other device handlers are also driven by the "tick" (communications, text window interface, voice-recogntion interface). TVCTRL is responsible for triggering each of these drivers in turn, as appropriate.

The Software Interface to TVCTRL

The main body of TVCTRL responds to the hardware "tick". In addition, each TVCTRL implementation includes an interrupt handler installed under INT 65H that provides a software interface by which other processes can interact with TVCTRL. The particular function performed by INT 65H depends on the value in the AH register. The INT 65H functions that apply to all versions of TVCTRL are defined as follows:

AH=0 Enable TVCTRL only (do not enable other handlers driven by TVCTRL)

AH=1 Disable TVCTRL only

(AH=2 Used by TVSSEC only; for passing messages to the "tower". See the section below on TVSSEC.)

AH=3 Return TVCTRL state. Value returned in AL:
- AL=0 TVCTRL only disabled
- AL=1 TVCTRL only enabled
- AL=6 TVCTRL and other handlers it drives disabled
- AL=7 TVCTRL and other handlers it drives enabled

AH=4 Initialize TVCTRL. Enable TVCTRL and other handlers driven by it. Enable keyboard filter.

AH=5 Completely disable TVCTRL. I.e. disable TVCTRL and other handlers driven by it and disable keyboard filter.

AH=6 Enable TVCTRL and other handlers driven by it.

AH=7 Disable TVCTRL and other handlers driven by it.

Note: To enable or disable other handlers driven by TVCTRL really means, in this context, to enable or disable TVCTRL's triggering of these handlers. Each of the individual handlers also has its own enable/disable mechanism as well.

Fortran-callable interfaces are provided for certain of these functions. Their calling sequences are:

AH = 0 -- CALL ENBTVC

AH = 1 -- CALL DSBTVC

AH = 4 -- CALL INITVC

AH = 5 -- CALL TVCOFF

All other interactions with TVCTRL are performed by reading/writing the appropriate values in SYSCOM.

<u>The TVCTRL Semaphore</u>

TVCTRL is not re-entrant, so is not permitted to interrupt itself. It uses a local data flag as a semaphore; if TVCTRL is entered and the flag shows TVCTRL is already running, the later instant of TVCTRL exits immediately. Even if TVCTRL were re-entrant, one would want to use this kind of semaphore mechanism since there is nothing to be gained by allowing multiple instances of TVCTRL to be active at once, and without at least some upper bound on the number of instances allowed one runs the risk of overflowing the stack.

TVSSEC -- TV Control for the "Tower"

TVSSEC's Tick Mechanism

In the case of a PC-McIDAS workstation that displays its images and graphics on an SSEC/Dataram "tower", the hardware interrupt used to generate the TVCTRL tick is a TV timing interrupt generated by the tower on every other vertical retrace, 30 times per second. The AT interrupt occurs on IRQ15. The first thing TVSSEC does is send an EOI (End-of-interrupt) code to both of the AT's 8259 interrupt controllers to re-enable hardware interrupts. (The 8259's are cascaded, with IRQ15 wired to the slave. That's why both have to be re-enabled.)

The AT communicates with the tower via an AT-bus-to-MultiBus hardware interface manufactured by Bit3 Corp. Besides providing bi-ported memory that resides in the address space of both busses (at segment 0D000H in the AT), the Bit3 interface allows interrupts on the MultiBus to generate AT bus interrupts. This is the means by which the tower's TV timing interrupt produces an interrupt on the AT.

How TVSSEC Communicates with the Tower

TVSSEC is a somewhat atypical instance of TVCTRL. As described in the chapter "A Few Hardware Considerations", the tower contains an 8085 microprocessor that executes code in ROM. The 8085 takes care of actually displaying a particular frame on the screen, for example. The AT causes the 8085 to perform a given function by sending to the 8085, across the Bit3 interface, a comm packet in the same protocol the host formerly used to communicate with the tower directly. There are many functions that in a more typical instance of TVCTRL would be handled by the AT directly but which in TVSSEC are handled by constructing a packet and passing it to the 8085.

Two buffers are used for passing packets between the AT and the 8085. Both live in the bi-ported Bit3 memory. One, the so-called "quick buffer", is reserved for high-priority messages. Since the 8085 does no lookahead in processing messages sent to it, there is the potential for messages defining basic terminal state to have to wait in line behind relatively slow messages that do things like load enhancement tables. The quick buffer is used to let basic terminal state messages get immediate service. (The other buffer will be called the "slow buffer" in the discussion that follows.)

Each of the two buffers is divided into two halves, so the 8085 can be building reply messages in one buffer half while it reads through the messages received in the other half. In other words, at any given moment one half is treated as a receive buffer, the other as a transmit buffer.

Use of the buffers is synchronized by several flags that also live in the bi-ported Bit3 memory. For each of the two buffers there is a flag that indicates which machine (AT or 8085) currently controls the buffer, and a flag indicating which of the two halves of the buffer is currently the receive buffer.

The dialogue is half-duplex. When either machine (AT and 8085) gains control of a buffer, it immediately processes the messages in the buffer, formulates a reply (a null message if no other message is pending), and relinquishes control of the buffer.

The AT processes only three types of messages from the 8085; there are other types sent by the 8085, but they are ignored. The three that are processed are:

- ID response (routing code 8AH): the 8085 responds to an ID request by sending workstation ID, number of image frames, and number of graphics frames. The AT ignores the ID, but stores the numbers of frames in SYSCOM.

- Raw joystick data (routing code 9AH): the 8085 sends joystick position data. The AT stores joystick position in SYSCOM.

- Terminal cursor state (routing code 70H): the 8085 sends cursor position and size data. Which of these data, if any, are stored in SYSCOM depends on the cursor link mode.

When the AT receives a message or messages in the quick buffer (e.g. raw joystick data is constantly coming in), it processes them and sends back three packets in the quick buffer:

- Image frame control (routing code 91H)

- Graphics frame control (routing code 92H)

- Primary cursor control (routing code 93H)

This keeps the tower up-to-date on which frame to display and where to position the cursor.

When the AT receives a message or messages in the slow buffer, it processes them and then checks to see if it has any messages buffered up from applications. Applications-generated messages (including host-generated messages passed through by the communications software) are stored in their own buffer (the applications message buffer) until it's the AT's turn to talk, at which time all pending applications messages are sent at once.

Specifics of the INT 65H Interface for TVSSEC

Applications that want to send a message to the 8085 pass the message to TVSSEC via the INT 65H interface, using the following register settings:

AH=2 Function code to send a message to 8085.

DS:SI -- Pointer to the packet to be sent.

CX -- Packet length. (Exclude ETX character, if any.)

The Fortran-callable interface to function AH=2 of INT 65H is:

CALL SENOUT (SENBUF)

where SENBUF is a character array, and the packet in SENBUF is terminated by an ETX character (ASCII code 3).

Full Buffers and Deadlocks

If the applications message buffer is already full, the INT 65H code loops until buffer space is available. Buffer space eventually becomes available because the loop is interrupted by TVSSEC on each tick, and TVSSEC bails the applications message buffer as soon as a message is received in the slow buffer from the 8085.

Some care must be taken to avoid deadlocks. An instructive example is provided by a bug that existed at one time but is now fixed. Recall that one of the functions of TVCTRL is to trigger the other tick-driven interrupts -- the communications driver, in particular. Recall also that TVCTRL uses a semaphore to block re-entry. Formerly, the call to the communications driver was contained within the scope of TVSSEC's semaphore.

This permitted a deadlock in the event that the communications driver called INT 65H to send a packet to the 8085 when the applications message buffer was already full. (This would happen only very occasionally, usually when an image or graphic was coming in from the host at the same time a local PC-McIDAS command was generating packets for the 8085.) INT 65H would go into a loop waiting for the buffer to be bailed. The comm driver, in turn, waited for INT 65H to complete. But since the comm driver was called from within the critical region of the semaphore, TVSSEC could not execute, so it never had an opportunity to bail the applications message buffer. Deadlock. The solution was to move the call to the comm driver out of the critical region of the semaphore.

TVEGA -- TV Control for the IBM EGA and VGA

Basic Differences from TVSSEC

When the PC-McIDAS workstation uses an IBM EGA (Enhanced Graphics Adapter) or VGA (Video Graphics Array) as the imagery/graphics display device, the situation is quite different from the one described in the previous section.

For one thing, there is no longer an 8085 microprocessor interposed between the AT and the display; the AT interacts directly with the EGA/VGA hardware. As a result, TVEGA must handle functions that TVSSEC can leave to the 8085 (e.g. drawing the cursor). Moreover, TVEGA's structure is quite different from TVSSEC's, since the kind of message buffering that is the heart of TVSSEC isn't relevant to TVEGA.

For another thing, the memory used for storing frames must reside within the AT itself, not in the tower. Frames are stored in extended memory. At the time of this writing 16 frames are allocated, but this number could just as well be made user-configurable.

TVEGA's Tick Mechanism

The "tick" mechanism used to drive TVEGA is the "time-of-day" interrupt INT 1CH. This interrupt occurs 18.2 times per second.

Specifics of the INT 65H Interface for TVEGA

TVEGA itself is installed under INT 1CH; in addition, as was the case with TVSSEC, there is a software interface installed under INT 65H. Besides the functions that apply to all instances of TVCTRL, INT 65H has the following special functions for TVEGA:

AH=0F0H	Erase the cursor before graphics plotting
AH=0F1H	Redraw the cursor after graphics plotting
AH=0F2H	Force the current frame to be re-displayed
AH=0F3H	Set a flag used to prevent TVEGA from updating the screen

AH=0F4H Clear the flag used to prevent TVEGA from updating the screen

AH=0F6H Draw the cursor on a blank screen

More will be said about these functions later on.

<u>How Frames Are Handled</u>

As mentioned above, frames are stored in extended memory. The set of data areas in extended memory used to store the frames will be referred to, collectively, as "frame space".

To display a frame means to move the frame's data from frame space (extended memory) to the memory of the graphics hardware. As described in the chapter "Using the IBM EGA and VGA", data in the graphics memory are organized in bit planes. For the sake of efficiency, therefore, the frame data is stored as bit planes in frame space as well.

Each frame has 350 lines and 640 pixels per line, making 224000 pixels in all. Each bit plane, therefore, requires 28000 bytes (224000 bits) of storage. There are 4 bit planes per frame, so each frame requires 4 * 28000 = 108000 bytes of storage. Frames are laid out contiguously in frame space, 108000 bytes per frame.

When a new frame is to be displayed, the frame is moved from frame space to the graphics memory one bit plane at a time. If the frame is moved into the part of graphics memory currently being displayed, this process causes the colors to flash as the various bit planes are loaded. Fortunately, there are 2 pages of graphics memory available. A frame is always loaded into the page which is **not** currently visible, then the page register is modified to bring that page into view. This latter process is essentially instantaneous, so no flash is visible.

For more details on how to load bit planes into the graphics memory, see the chapter "Using the IBM EGA and VGA".

Frame Numbers and the Cursor

After a frame is displayed, the frame number is drawn in the lower left corner of the screen. The number is drawn by TVEGA, rather than by the EG command as is done in tower-based workstations, because EGA/VGA-based PC-McIDAS supports saving images to files. One does not want the frame number to be written into an image that is going to be saved to a file and possibly restored later to a different frame.

Each time a new frame is displayed (e.g. when looping) the frame number must be drawn. TVEGA handles the frame number drawing directly for maximum performance. That is, it generates the digit characters itself and writes directly to the EGA/VGA hardware, by-passing BIOS.

The cursor is also drawn by TVEGA directly. It is drawn only if it has changed or moved, or if a new frame has been displayed. In other words, it is drawn only as needed.

Each time the cursor is drawn, it is necessary first to read and save the values of all the pixels that will be over-written by the cursor so they can be restored when the cursor is moved. TVEGA has a buffer big enough to support the largest possible box with cross hairs. TVEGA refuses to draw a solid cursor because it cannot afford to buffer enough data to support a large solid cursor.

Drawing a moved cursor is a 3-stage process. First, the pixels that were over-written the last time the cursor was drawn have to be restored. Then the pixels that are about to be over-written have to be saved. Finally, the cursor is drawn in its new position. To get smooth cursor movement, it is essential that this process be handled quickly. TVEGA accesses the EGA/VGA hardware directly, which is much faster than using BIOS calls.

In a tower-based workstation, the cursor resides conceptually in its own overlay, distinct from the image frame and the graphics overlay. Images and graphics can be drawn without worrying about what they might do to the cursor.

An EGA/VGA-based workstation, however, presents new difficulties. Suppose, for example, that a cursor is drawn on a blank (black) frame, an image is loaded, and the cursor is then moved. When the cursor is moved, the pixels over-written by the cursor will be restored to the values they had when the cursor was drawn. But the frame was blank then, so a black "ghost" of the cursor gets drawn into the image. Analogous problems occur if a graphic is drawn through a cursor.

Such difficulties are most easily overcome by erasing the cursor before loading an image or drawing a graphic and then restoring it after. The plot package for PC-McIDAS does just that. The cursor is erased (by calling INT 65H with AH=0F0H; see above) when INITPL is called; it is restored (by calling INT 65H with AH=0F1H) when ENDPLT is called.

Similarly, when a frame is erased, the new cursor is drawn (by calling INT 65H with AH=0F6H) without first restoring over-written pixels. Otherwise, one would get a "ghost" cursor consisting of whatever pixels were under the cursor when the frame was erased.

COMMUNICATIONS DRIVERS

Overview

PC-McIDAS is designed to support a variety of communications links between the workstation and a host computer. Currently supported are the following:

- ProNET local area network.
- Asynchronous serial comm, up to 19.2 KBaud, via telephone dial-in, direct line, or satellite broadcast.

Alternatively, a workstation may be configured to stand alone.

Which of these modes is currently in use in a workstation is indicated by the value of Byte 383 of the Terminal Control Block (TCB) of SYSCOM.

The comm drivers are installed as interrupt handlers rather than linked into other systems or applications software. This makes it possible to change a workstation's comm mode or implement a new mode without modifying other software. Only the comm drivers need to be re-installed.

In each case (ProNET and async) there are two levels of drivers: a low-level driver responsible for interacting with the comm hardware to receive/transmit packets, and a high-level driver responsible for interpreting and acting upon packets. These drivers are named as follows:

Comm Mode	Low-level Driver	High-level Driver
ProNET	PNETINT	COMMP
Async	ASYNC1, ASYNC2	COMMA
Standalone	---	COMMN

Each of these drivers is described in more detail below.

PNETINT -- The Low-Level ProNET Comm Driver

The low-level ProNET driver PNETINT is installed under INT 61H. PNETINT itself is written in C. It interacts with the ProNET board to receive and transmit packets. Much of the very lowest level activity is handled by the ProNET board itself, transparently to the PC. The particular function performed by a call to INT 61H depends on the value in the AH register as follows:

```
AH=0  --  Enable receiver.
          Input:     none
          Output:    AX = status

AH=1  --  Check receiver.
          Input:     ES:BX = pointer to message buffer
          Output:    AX = status
                     BX = extended status
                     CX = message length
                     DX = node of sender

AH=2  --  Enable transmitter.
          Input:     ES:BX = pointer to message buffer
                     CX = message length
                     DX = destination node

AH=3  --  Check transmitter.
          Input:     none
          Output:    AX = status
                     BX = extended status

AH=4  --  Correct last receive error.
          Input:     none
          Output:    AX = status

AH=5  --  Correct last transmit error.
          Input:     none
          Output:    AX = status

AH=6  --  Reset and connect to ring.
          Input:     none
          Output:    AX = status

AH=7  --  Disconnect from ring.
          Input:     none
          Output:    AX = status
```

The following status codes are defined:

0 -- operation done/successful
1 -- correctible packet error (1-bit protocol, parity, etc.)
2 -- ProNET hardware failure
80H -- operation in progress/wait
FFH -- unrecognized function

The extended status is defined as follows:

```
BH = ???? ??ER
     E = current one-bit protocol state
     R = one-bit protocol state received
           (after receive only)
BL = ProNET receive/transmit control status register
```

Various Fortran-callable functions exist to activate these functions (see PINTLNK.ASM). Each returns a status. They are, respectively:

INTEGER FUNCTION **ENBRCV**

INTEGER FUNCTION **CHKRCV**(BUFFER,LENGTH,NODE)

INTEGER FUNCTION **CORRCV**

INTEGER FUNCTION **ENBXMT**(BUFFER,LENGTH,NODE)

INTEGER FUNCTION **CHKXMT**

INTEGER FUNCTION **CORXMT**

INTEGER FUNCTION **COMRST**

INTEGER FUNCTION **COMDSB**

To receive a message, call ENBRCV, then call CHKRCV repeatedly until a message is found. To transmit a message, call CHKXMT repeatedly until the previous transmit has completed, then call ENBXMT. Note, however, that applications programs **never** call the above routines. Rather, they are called by COMMP, the high-level ProNET comm driver.

ProNET workstations maintain a strictly half-duplex dialogue with the host. The one-bit protocol is a device for detecting and correcting simple errors in this dialogue. Under the protocol, the workstation and the host each toggle between two states. Each state expects an incoming message to include a predetermined state ID (0 or 1). If an incorrect ID is received, a comm error is indicated (e.g. a lost packet) and the receiver resends the last message it sent. For a detailed description, see the document "Host to Terminal - Terminal to Host System Protocol Description" (which describes the so-called 02/03 protocol).

In particular, the one-bit protocol enables the workstation to force retransmission of a packet received in error. All it has to do is retransmit the last message. CORRCV and CORXMT both do the same thing: retransmit the last message.

COMMP -- The High-Level ProNET Comm Driver

PNETINT, the low-level driver, takes care of receiving and transmitting packets. Interpreting and acting on the packets is handled by COMMP, the high-level driver (source code in PCOMM.ASM).

COMMP is installed under INT 64H. It is triggered on each tick by TVCTRL. The triggering of COMMP can be inhibited by setting the flag in Byte 387 of the TCB.

PCOMM.ASM is a long program, but the structure is fairly straightforward. It is probably worthwhile to point out a few landmarks, however.

COMMP is not re-entrant, but it uses a local semaphore to fend off re-entry. Thus, there is no problem with triggering INT 64H from processes other than TVCTRL.

Since the dialogue with the host is half-duplex, the flow of control through COMMP depends heavily on the state of a local flag (MYTURN) that indicates whether it is the workstation's turn to talk or the host's. If it is the host's turn to talk, COMMP calls a procedure named RECEIVE; otherwise, it calls TRANSMIT.

RECEIVE calls CHKRCV to see if a packet has come in. If so, it calls a procedure DOMSG that parses the packet to extract the logical packets it contains. For each logical packet, DOMSG jumps to the section of code appropriate to the particular routing code in question. If CHKRCV indicates that no packet has come in, RECEIVE exits. In that case, the MYTURN flag does not change, so RECEIVE gets called again on the next tick. It will continue to get called on each tick until a packet is received or a timeout condition arises.

If a received packet cannot be processed in the same tick in which it was received (e.g. if it contains data that must be saved in a disk file, but a foreground process is already using a DOS function; or, e.g., if it contains CRT text but the CRT text buffer is full because a Ctrl-S is in effect) a flag (HELDOVER) is set and RECEIVE exits. On the next tick, MYTURN is still clear so RECEIVE gets called again. Since HELDOVER is set, RECEIVE skips the call to CHKRCV and treats the heldover packet as if it just came in. This process is repeated until COMMP is able to process the packet, at which time HELDOVER is cleared.

Note that the workstation does not send or receive any new packets while HELDOVER is set. The comm dialogue is suspended. The usual case is a packet being held over until a foreground DOS function completes. Here, the interval is usually just a few ticks, so it is imperceptible. There are cases, however, in which the suspension of the comm dialogue is apparent to the user. For example, if a user enters a Ctrl-S when a lot of text is coming down from the host, COMMP's CRT text buffers will fill and the comm dialogue will cease. This has several minor side-effects. The LED's that signal ProNET activity will stop blinking, and when the Ctrl-S is countermanded there will be a short pause before the host realizes the workstation has resumed the dialogue.

When it is the workstation's turn to talk, the situation is a little more complicated. Messages to be transmitted can arise either at the applications/scanner level or at the level of the comm software itself. At the comm software level there are two main cases:

- Various kinds of packets from the host are requests for data concerning the state of the workstation. COMMP takes care of constructing and sending the needed reply packets, getting the required data from SYSCOM. Certain kinds of replies are sent only after a specified delay, so there is a data structure that stores each pending reply routing code together with a delay count. The delay counts are decremented on each tick; a reply packet is constructed for each delay count that reaches 0 on a given tick.

- In the absence of other traffic, the workstation and host exhange "idle" packets. Whenever the workstation transmits, the host responds immediately. It is the workstation's responsibility, therefore, to insert a delay before sending an idle packet. The delay "ramps up" to about 2 seconds when nothing else is going on.

Packets to be transmitted may also be generated by the PC-McIDAS command scanner or by applications programs. In such cases, the process generating the packet leaves mail in SYSCOM. The TRANSMIT procedure in COMMP checks to see if mail is waiting; if so, it takes care of transmitting the packet. A status is returned in SYSCOM for the generating process.

Note that a number of logical packets may be generated on a single tick. It is desirable to send these in as few physical packets as possible. COMMP has a procedure CHKPACK that takes care of building up a physical packet, separating logical packets with inter-record separator (IRS) characters, and transmitting the physical packet when its buffer becomes full. CHKPACK calls ENBXMT to send the packet, then calls CHKXMT in a loop until the packet has actually been sent. If CHKXMT returns an error status, CHKPACK takes care of error-handling.

One comment needs to be made about error-handling. If an error persists after a few retries, a short delay is inserted before each subsequent retry. This delay = 20 msecs + (2 msecs * workstation's node address). The purpose of this computation is to produce a different delay for each workstation on a given ProNET ring. This is essential to prevent a dynamic deadlock when the ring token is lost. Without it, all workstations simultaneously attempt to reset the ring and the token keeps getting eaten.

ASYNC -- The Low-Level Async Comm Driver

There are two versions of the low-level async comm driver: ASYNC1 and ASYNC2. The only difference between them is which serial port they use. They will referred to collectively as ASYNC.

Having two versions of ASYNC is a somewhat clumsy way to handle the problem of two possible serial ports. The problem is that ASYNC must know at install-time which port it is going to use, but the port number is not initialized in SYSCOM until MCIDAS run-time. A unified version of ASYNC could be created, however, and probably should be. For example, one could write a little program that accesses the file \MCIDAS\SETUP\CONFIG.DAT to determine which serial port is going to be used. This program could be run after SYSCOM is installed but before ASYNC is installed and could initialize the appropriate SYSCOM value.

Like PNETINT, ASYNC is installed under INT 61H and provides, under INT 61H, various functions for sending and receiving packets, etc. ASYNC differs from PNETINT, however, in that ASYNC also installs code to respond to hardware interrupts at the byte level. (In the ProNET case, the byte level processing is handled by the ProNET board.)

The functions performed by INT 61H depend on the value in the AH register, as follows:

```
AH=0  --  Initialize ASYNC.

AH=1  --  Disable ASYNC.

AH=2  --  Receive a packet.
          Input:    ES:DI = pointer to message buffer
          Output:   AX = status
                    CX = message length

AH=3  --  Receive data unconditionally.
          Input:    ES:DI = pointer to message buffer
          Output:   AX = status
                    CX = message length

AH=4  --  Transmit a packet.
          Input:    DS:SI = pointer to message buffer
                    CX = message length
          Output:   AX = status

AH=8  --  Send an XOFF.
          Input:    none
          Output:   none
```

AH=9 -- Send an XON.
Input: none
Output: none

AH=10 -- Send an XOFF and wait for it to take effect.
Input: none
Output: none

(The functions for AH=3 and AH=8 are not used by PC-McIDAS.)

The following status codes are defined:

0 -- operation done/successful
1 -- data overflow/data lost
80H -- operation in progress/wait
FFFFH -- unrecognized function

Certain byte values are interpreted by the host's controller firmware as control characters, so they are converted to escape sequences. The values that must be escaped are: 8, 13, 17, 19, 26, 27, 145, 147. Any value from this list is converted to an ESC followed by the value OR'ed with 60H. On input, therefore, all ESC characters are dropped and each character that followed an ESC is AND'ed with 90H.

Escape sequences aside, ASYNC assumes all incoming data are either packets that conform to the F0-protocol or else are pure ASCII text.

The remainder of this section outlines the structure of the ASYNC source code.

The hardware interrupt entry point is ASYINT. It does an IN instruction to get the value of the serial interrupt ID register. If the ID value indicates an interrupt for data received, the procedure RCVINT is called to handle the received byte. If the ID value indicates an interrupt for transmit holding register empty, XMTINT is called to transmit the next byte. Note that ASYINT must go back and check the ID register again before it exits. It keeps iterating until the ID value is clear -- another interrupt may have been received while the first interrupt was being processed. Also, ASYINT always gives precedence to receive interrupts.

RCVINT simply buffers data as it comes in. It pays no attention to packet boundaries, nor does it de-escape ESC sequences. The "AH=2 -- Receive Packet" function of INT 61H scans through the input buffer to determine if a full packet has been received. If so, it returns the de-escaped packet and modifies the buffer pointers.

Transmission of packets is handled as follows. The "AH=4 -- Transmit Packet" function of INT 61H moves the packet to a buffer available to XMTINT, adding ESC sequences as appropriate. It then enables interrupt on transmit holding register empty. XMTINT is triggered by the interrupt repeatedly, sending a byte at a time, until the buffer is emptied. Note that INT 61H does not wait for the transmission to complete. It just loads the buffer, enables the interrupt, and exits. The actual transmission takes place asynchronously under interrupt control.

If INT 61H, AH=4 is called to transmit a packet while another packet is in the process of being transmitted, it simply exits, returning a busy status. It is up to the caller to retry later.

The INT 61H, AH=10 -- "Send XOFF and Delay" function (procedure ASXOFD) requires a little explanation. Ordinarily, a process sending an XOFF does not want to continue until the XOFF has actually taken effect and no more input data is being received. ASXOFD sends an XOFF and waits for an interval that depends on the baud rate. If no character comes in during that interval, it returns. Otherwise, it sends another XOFF and waits again, and so on. Moreover, when the last byte sent by ASYNC was an XOFF, every incoming byte is immediately answered with an XOFF.

Each time INT 61H is called to send an XOFF, it increments a counter; each time it is called to send an XON, it decrements the counter. It only actually sends the XON if the counter is back to 0. This way, XOFF/XON pairs may be nested without intermediate XON's getting sent and prematurely restarting data transmission by the host. Not counted in this way are XOFF's generated by ASYNC itself when its buffer gets nearly full nor XON's sent when the buffer later empties out, nor XOFF's generated by ASYNC when a byte is received after a prior XOFF. Note that the sending of an internally-generated XON (when the previously full buffer empties) is suppressed if the XOFF/XON count is nonzero.

XOFF/XON pacing is needed for another purpose besides preventing buffer overflow. Without it, serial data are lost when the workstation accesses extended memory. In order to access extended memory, the 80286 microprocessor must be switched into protected mode. Interrupts must be disabled while the processor is in protected mode since only interrupt handlers written for real mode are installed. Whenever interrupts are disabled for a long interval, serial data will be lost.

COMMA -- The High-Level Async Comm Driver

COMMA (source code in ACOMM.ASM) is structured somewhat like COMMP. COMMA has procedures named RECEIVE, DOMSG, TRANSMIT, DSPCRT, etc. that function analogously to the procedures with those names in COMMP. There are important differences between COMMA and COMMP, however.

The async workstation-host dialogue is full duplex, not half duplex as in the ProNET case. COMMA calls RECEIVE on each tick on which it has nothing to transmit. Note, however, that there is a flag in SYSCOM (Byte 396 of TCB) that blocks COMMA from calling RECEIVE. This flag is used by PC-McIDAS commands like GETPRD and FONHOM that need to intercept all incoming data to check for the replies indicating successful dial-in. By blocking calls to RECEIVE they ensure that COMMA does not get the data before they do.

Other differences: COMMA expects F0-packets or pure text, and there is no "idle" dialogue in an async connection.

Async workstations also handle temporary files differently. Temp files are used to store incoming LW-file packets. Care must be taken in opening and closing such files. Suppose a temp file is opened while a PC-McIDAS command is running. When the command completes, DOS closes all files that were opened while the command was running, whether or not it was the command that opened them. DOS expects only one process at a time to be using files. The temp file may be closed prematurely.

In the ProNET case, LW-file transfers are infrequent enough that it is sufficient to hold off opening a temp file if a PC-McIDAS command is currently running. In the async case, however, particularly with broadcast reception (UNIDATA) workstations, LW-file transfers are happening all the time. The solution in the async case has been to maintain a pool of 5 temp files that are re-used over and over. All 5 files are opened during PC-McIDAS initialization, when no commands are running. They are kept open until PC-McIDAS is exitted.

Special Requirements for Broadcast Reception (e.g. UNIDATA)

As noted above, XOFF/XON pacing is needed to keep async workstations from dropping data when the workstation accesses extended memory. When the workstation is receiving a satellite broadcast, however, it is impossible to pace the host.

In this instance, a "buffer box" is interposed between the broadcast reception hardware and the workstation. The buffer box allows serial in and serial out, contains a 256KB buffer, and responds to XOFF/XON pacing. The workstation gets its data from the buffer box and is able to pace it.

Since the buffer box continues to fill while it is X'ed OFF, the workstation must be able to bail the buffer faster than the broadcast is filling it or else the buffer will overflow. ASYNC is set up to receive at 19.2 KBaud in this case; the satellite broadcast is at 9.6 KBaud. 19.2 KBaud is not supported by BIOS. See the ASYNC source code for how to get around this limitation.

THE TEXT WINDOW INTERFACE

Introduction

The text window interface allows text output to be parked in any of 10 virtual windows. Any window can be instantly brought to the screen at any time. Text can be written in color, positioned on the screen, caused to blink. Windows can be scrolled up or down.

These are desirable features, but their implementation poses certain problems. In particular, the existence of the text window interface must be transparent to non-PC-McIDAS programs like DOS. Moreover, the windows themselves must be stored and controlled in a way that gives instantaneous response and does not steal RAM needed by PC-McIDAS applications.

To make the text windows interface transparent to non-PC-McIDAS programs, it was necessary to implement text reading and writing at the BIOS level, rather than at the PC-McIDAS applications level. For this purpose, the BIOS video interrupt INT 10H was replaced by a new interrupt handler, VIDEO. Moreover, VIDEO had to be re-entrant.

To keep from without wasting RAM needed by applications programs, the text windows were stored in extended memory. To permit instantaneous response, the windows are controlled by a resident interrupt handler, SCREENS that is triggered on every "tick". SCREENS is responsible for displaying a window's text on the screen, for switching windows, scrolling them, etc.

Memory Usage

Memory is reserved for 10 text windows followed by 10 soft tablet windows. These 20 windows are laid out contiguously in extended memory starting at address 200000H (2 megabytes).

When the AT/PS2 is in real mode, address line 20 is masked off. The AT designers elected to do this as a cobble to rescue existing software that relied on wraparound of addresses above 1 megabyte. Unfortunately, it makes it impossible to use an in-circuit debugger to view memory in the 1-2 megabyte range when the 80286 is in real mode. This is a strong motivation for putting windows and EGA/VGA frames at addresses starting at 2 megabytes. The memory from 1-2 megabytes is reserved for a RAM disk in which is stored the pull-down menu HELP interface.

Each text window has 40 rows, only 23 of which are actually displayed at any one time. The number 40 could be increased, though there are performance tradeoffs. The larger the number of rows, the more work is involved in scrolling the screen. Each text window is allocated 6406 bytes (1910H), as follows:

Bytes	Use
0-1	Row number (0-based) for top row displayed
2-3	Cursor row number (0-based)
4-5	Cursor column number (0-based)
6-6405	Text data (2 bytes per character: ASCII code and attribute)

Each soft tablet window is allocated 4000 bytes (0FAH); 25 rows * 80 columns * 2 bytes per character.

In addition, a 6406 byte work area is reserved in a local data segment by SCREENS (in real-mode, i.e. non-extended, memory). This work area contains the currently displayed window. The work area is what actually gets displayed on the screen. Any operation that modifies the currently displayed window modifies the work area only. The work area contents are not stored in extended memory until the user changes to a different window. This architecture is necessary because accesses to extended memory are extremely slow compared to accesses to real-mode memory.

When SCREENS is initialized, it stores in SYSCOM the segment address of its work area. This is done to allow VIDEO also to have access to the work area.

VIDEO -- The BIOS INT 10H Replacement

The BIOS INT 10H video interrupt is re-vectored to INT 62H. Applications that need, for some reason, to call the BIOS interrupt code directly may do so by triggering INT 62H.

INT 10H is taken over by a PC-McIDAS module, VIDEO.EXE. Various INT 10H functions that write text to the screen, move the cursor, etc. are handled by VIDEO itself. Certain other functions, such as setting the graphics mode, are simply passed through to INT 62H. Functions handled directly by VIDEO include:

Function	Description
AH=2	Set cursor position
AH=3	Read cursor position
AH=6	Scroll page up (Clear screen ONLY)
AH=7	Scroll page down
AH=8	Read char and attribute
AH=9	Write char and attribute
AH=0AH	Write char
AH=0EH	Write TTY

The above functions are defined as for BIOS INT 10H. For VIDEO, two additional functions are defined:

AH=13H AL=0FFH	Write string (String may contain multiple lines; i.e. embedded CR/LF's are ok) ES:DI == pointer to string CX == length BL == attribute
AH=03FH	Write TTY DL == char to write

One of the requirements for VIDEO is that it must be able to handle calls from non-PC-McIDAS programs. This means that the window number must be passed through SYSCOM rather than as a parameter in a register. Byte 6 of the User Interface Block (UIB) contains the window number used by VIDEO. (Byte 5 of the UIB, by the way, contains the number of the window currently in SCREEN's work area.)

A call from a non-PC-McIDAS program automatically uses the current window number, since such a program doesn't know to set the window number in SYSCOM, but PC-McIDAS programs can modify the window number if desired to write to a non-displayed window. Processes (e.g. COMM) that call VIDEO from the background must save the current SYSCOM value before they modify it and call VIDEO, and they must restore it when control returns from VIDEO.

Calls to VIDEO that need to access the currently displayed window act on SCREENS' work area. Those that access another window act directly on the contents of extended memory. If text is written to a window other than window 0, the default condition is that the text is written to window 0 as well. In that instance VIDEO must modify both the work area and a window in extended memory -- or if neither window is in the work area, VIDEO must modify two windows in extended memory.

One of the requirements of VIDEO is that it should be re-entrant. For this reason, it must store local data on the stack. One situation where this becomes an important consideration is in connection with scrolling a window.

Generally speaking, each time a line of text is written the window must be scrolled one line. This means moving the entire text contents of the window. One must read the contents of the window and write it back, shifted one line. The text that is read must be stored on the stack. The need to store the text on the stack motivates one to read as few lines at a time as possible. Performance considerations, however, motivate one to read as many lines at a time as possible, since this minimizes the number of accesses to extended memory.

The current compromise is to allocate 40 lines per window and scroll 10 lines at a time. When VIDEO is modifying both the work area and a window in extended memory, scrolling the window in extended memory causes a noticeable slowdown in text writing. Increasing the number of rows per window beyond 40 would worsen performance in this regard. Some experimentation would be worthwhile, however, since it would be nice to have access to more than 40 lines per window.

It was necessary to make VIDEO re-entrant because it can be called freely by unknown, non-PC-McIDAS processes. It is also necessary, however, to limit the ways in which it by VIDEO. (Byte 5 of the UIB, by the way, contains the number of the window currently in SCREEN's work area.)

A call from a non-PC-McIDAS program automatically uses the current window number, since such a program doesn't know to set the window number in SYSCOM, but PC-McIDAS programs can modify the window number if desired to write to a non-displayed window. Processes (e.g. COMM) that call VIDEO from the background must save the current SYSCOM value before they modify it and call VIDEO, and they must restore it when control returns from VIDEO.

Calls to VIDEO that need to access the currently displayed window act on SCREENS' work area. Those that access another window act directly on the contents of extended memory. If text is written to a window other than window 0, the default condition is that the text is written to window 0 as well. In that instance VIDEO must modify both the work area and a window in extended memory -- or if neither window is in the work area, VIDEO must modify two windows in extended memory.

One of the requirements of VIDEO is that it should be re-entrant. For this reason, it must store local data on the stack. One situation where this becomes an important consideration is in connection with scrolling a window.

Generally speaking, each time a line of text is written the window must be scrolled one line. This means moving the entire text contents of the window. One must read the contents of the window and write it back, shifted one line. The text that is read must be stored on the stack. The need to store the text on the stack motivates one to read as few lines at a time as possible. Performance considerations, however, motivate one to read as many lines at a time as possible, since this minimizes the number of accesses to extended memory.

The current compromise is to allocate 40 lines per window and scroll 10 lines at a time. When VIDEO is modifying both the work area and a window in extended memory, scrolling the window in extended memory causes a noticeable slowdown in text writing. Increasing the number of rows per window beyond 40 would worsen performance in this regard. Some experimentation would be worthwhile, however, since it would be nice to have access to more than 40 lines per window.

It was necessary to make VIDEO re-entrant because it can be called freely by unknown, non-PC-McIDAS processes. It is also necessary, however, to limit the ways in which it can be interrupted by the PC-McIDAS background processes -- SCREENS and COMM -- that potentially switch the currently displayed window. The danger is that VIDEO may be in the middle of writing to the work area, for example, when it is interrupted by a process that moves a different window into the work area. When control returns to VIDEO it would then be writing into the wrong window. To handle this situation, there is a semaphore (byte 374 in the TCB) that lets VIDEO prevent SCREENS and COMM from switching windows.

SCREENS -- The Text Window Handler

Conceptually, there are two distinct aspects to SCREENS. In one aspect SCREENS is tick-driven; in the other it responds to calls from other processes. (This is similar conceptually to the situation with TVCTRL, which is tick-driven but also has its INT 65H interface through which it can be called by other processes.)

SCREENS is installed under INT 66H. The function performed by INT 66H depends on the value in register AX. The AH values used may seem weird at first glance, but they are the scan codes for the keypad keys that govern the window interface functions. When a keypad key is struck, TVCTRL just passes to INT 66H the ASCII code (AL) and scan code (AH) of the key. Naturally, any process can produce the same effect that a keypad key does if the process sets AL and AH appropriately and triggers INT 66H. On any tick on which no keypad key is found, TVCTRL calls INT 66H with AX=0.

Certain of the AH values correspond to two different possible functions. This results from the fact that certain scan codes are associated with two different ASCII codes, depending on whether NUM LOCK is on. If NUM LOCK is on, the ASCII code (AL) will be 0. The INT 66H functions are:

AH=0	Tick-driven call. Update the screen.
AH=71	Switch to window 7.
AH=72	If AL=0: Switch to window 8. If AL>0: Scroll up one line.
AH=73	If AL=0: Switch to window 9. If AL>0: Page up.
AH=75	Switch to window 4.
AH=76	Switch to window 5.
AH=77	Switch to window 6.
AH=78	Toggle among text windows, soft tablet, EGA/VGA frames.
AH=79	Switch to window 1.
AH=80	If AL=0: Switch to window 2. If AL>0: Scroll down one line.

AH=81	If AL=0: Switch to window 3. If AL>0: Page down.
AH=96	Force EGA/VGA frame to screen.
AH=97	Force soft tablet to screen. (AL=tablet num)
AH=98	Echo keyin only. Do not refresh entire screen.
AH=99	Force text window to screen. (AL=window num)
AH=255	Initialize.

The functions for AH=0, 96, 97, 98, 99, and 255 are activated by other processes. They do not correspond to keypad scan codes.

A Fortran-callable entry point exists to allow other processes to trigger the INT 66H. The calling sequence is:

```
CALL WNDINT (AH_REGISTER,AL_REGISTER)
```

Thus, for example, to force text window 5 to the screen a Fortran program would execute the following statement:

```
CALL WNDINT (99,5)
```

When the text windows are visible, SCREENS ordinarily updates the screen from its internal work area on each tick.

The AH=98 function is called by the scanner when echoing the command line. It causes the command line only to updated.

GRAPHICS DRIVERS

Introduction

Various PC-McIDAS applications programs need to draw image or graphics pixels in an image/graphics frame. It is desirable, however, for the applications programs themselves to be independent of the particular display hardware being used. To achieve this device independence, low-level image/graphics entry points are implemented via a software interrupt. The code that actually interfaces to the display hardware is installed as a resident interrupt handler (INT 63H), so it is not linked into any applications program. To change to a different display device one simply installs the resident driver appropriate to that device. Applications programs do not change at all.

The generic name for the image/graphics interface driver is PV. The version specific to the SSEC-Dataram-Conrac "tower" display is called PVSSEC. The version for the IBM EGA/VGA is called PVEGA.

The specific function performed by PV (INT 63H) depends on the setting of the AL register, as follows:

AL=0 -- Set graphics window

AL=1 -- Draw graphics point

AL=2 -- Draw graphics line segment

AL=3 -- Load tv image line

AL=4 -- Initialize the driver

Fortran-callable entry points are available to enable applications programs to activate the various PV functions. They are, respectively:

AL=0 -- CALL GRWNDW (UPLLIN,UPLELE,LWRLIN,LWRELE)

UPLLIN == Upper left line (0-based)
UPLELE == Upper left element (0-based)
LWRLIN == Lower right line (0-based)
LWRELE == Lower right element (0-based)

```
AL=1 -- CALL P (FRAME,LINE,ELEMENT,COLOR,IFLAG)

       FRAME   == Frame number (1-based)
       LINE    == Line number (0-based)
       ELEMENT== Element number (0-based)
       COLOR   == Graphics color (Device dependent)
       IFLAG   == At the present time, IFLAG should = 1

AL=2 -- CALL GRLINE (FRAME,COLOR,BEGLIN,BEGELE,ENDLIN,
            ENDELE,WIDTH,DSHLEN,GAPLEN,GAPVAL)

       FRAME   == Frame number (1-based)
       COLOR   == Graphics color (Device dependent)
       BEGLIN  == Line for beginning pixel in segment
                  (0-based)
       BEGELE  == Element for beginning pixel in segment
                  (0-based)
       ENDLIN  == Line for ending pixel in segment
                  (0-based)
       ENDELE  == Element for ending pixel in segment
                  (0-based)
       WIDTH   == Segment width in pixels
       DSHLEN  == Dash length in pixels (0 for no dashing)
       GAPLEN  == Gap length in pixels (0 for no dashing)
       GAPVAL  == Gap color (device dependent)
```

(GRLINE is not called by applications programs directly. See the discussion below of subroutine DRWLIN.)

```
AL=3 -- CALL V (FRAME,LINE,PIXEL,NUMPIX,IARRAY,IPLOT)

       FRAME   == Frame number (1-based)
       LINE    == Line number (0-based)
       PIXEL   == Starting pixel number (0-based)
       NUMPIX  == Number of pixels to be loaded
       IARRAY  == Array containing pixel values
       IFLAG   == At the present time, IFLAG should = 1

AL=4 -- CALL PVINIT
```

(By the way, the "PV" nomenclature derives from the entry points P and V, above.)

The PV interrupt handlers (i.e. PVSSEC and PVEGA) are intended to be called via the entry points listed above. Accordingly, they are set up to take their parameters from the stack. Generally speaking (there are certain exceptions, see below), each of the entry points GRWNDW, P, GRLINE, V, and PVINIT simply sets the AL register to the appropriate value and performs the INT 63H instruction. The stack is unaffected, except that 6 bytes (flags register and far return address) are pushed by the INT 63H instruction itself. PV (INT 63H) extracts its parameters from the stack, allowing for the extra 6 bytes.

It is intended, moreover, that PV is called only from the foreground. Background drivers (e.g. TVCTRL) that need to write to an image/graphics frame do so directly. In particular, no attempt is made either to make PV re-entrant or to serialize access to it.

The preceding remarks pertain to all versions of PV (i.e. PVSSEC, PVEGA, and any future implementation). The following two sections will describe considerations that pertain specifically to PVSSEC and PVEGA.

In addition to the above entry points, PC-McIDAS supports the entry points in the usual McIDAS plot package -- INITPL, PLOT, and ENDPLT, in particular. The PC-McIDAS plot package is discussed in the third section below.

PVSSEC -- PV for the Tower

As is described more fully elsewhere (see the chapters "A Few Hardware Considerations" and "TV Control"), PC-McIDAS workstations that use the SSEC/Dataram "tower" are controlled by having the AT formulate comm packets that are passed to the 8085 in the tower. To pass such a packet to the 8085, a program invokes INT 65H.

To support the implementation of PVSSEC, four new routing codes have been added to the so-called 02/03 protocol used for AT-to-tower communications. These new routing codes support 8-bit data. They are defined as follows:

Routing code 60H -- Set graphics window (default is full screen).

```
Bytes 1-2  -- Upper left element (0-based)
Bytes 3-4  -- Upper left line (0-based)
Bytes 5-6  -- Lower right element (0-based)
Bytes 7-8  -- Lower right line (0-based)
```

Routing code 61H -- Graphics point draw (draws only points that are within the current window).

Byte 1	-- Graphics frame number
Byte 2	-- Graphics color
Bytes 3-4	-- Element (0-based)
Bytes 5-6	-- Line (0-based)

Routing code 62H -- Graphics line segment draw (draws only points that are within the current window).

Byte 1	-- Graphics frame number
Byte 2	-- Graphics color
Bytes 3-4	-- Starting element (0-based)
Bytes 5-6	-- Starting line (0-based)
Bytes 7-8	-- Ending element (0-based)
Bytes 9-10	-- Ending line (0-based)
Byte 11	-- Width of segment in pixels
Byte 12	-- Dash length (0 for no dashing)
Byte 13	-- Gap length (0 for no dashing)
Byte 14	-- Gap color

Routing code 63H -- TV image line load

Byte 1	-- Image frame number
Bytes 2-3	-- Starting pixel within line (0-based)
Bytes 4-5	-- Line number (0-based)
Bytes 6-7	-- Number of pixels (max=number of pixels in a full line)
Bytes 8-N	-- Pixel values (8 bits per pixel)

In the case of PVSSEC, the initialization function (AL=4) is a no-op. It is present purely for symmetry with PVEGA, where it is needed.

PVEGA -- PV for the EGA and VGA

A certain amount of the functionality in PVEGA is now obsolete. At one time, PC-McIDAS on the EGA supported use of graphics modes 4 and 6 with a variable number of image/graphics frames stored in lower memory. There was a PC-McIDAS command called EGA that let the user change the number of frames or the graphics mode. PVEGA still supports this functionality.

When used in graphics mode 4 or 6, PVEGA needs quick access to frames' segment addresses and offsets of lines within a frame. These values are stored in tables for quick lookup. The tables are updated by PINIT and SETP whenever the graphics mode is changed (which nevers happens in the current implementation). It would make sense at some point to extract these tables and the accompanying functionality from PVEGA, PINIT, SETP, and GRINIT (GRINIT is described below). This would save 1-2 KB of memory.

The IFLAG parameter used by the P and V entry points (see the "Introduction" to this chapter) also reflects the functionality in graphics modes 4 and 6. It used to be possible to write to a frame not currently displayed and to draw pixels by XOR-ing. IFLAG allowed one to select the mode of drawing.

There is a program called GRINIT that is spawned to initialize the graphics subsystem when MCIDAS.EXE is started up. Under the old functionality, GRINIT was a subroutine called from MCIDAS. GRINIT needed to be linked into MCIDAS because it allocated memory used for the image/graphics frames. It could not be a separate program, or its memory would be freed as soon as GRINIT exitted. The EGA command that allowed the user to change the number of frames or the graphics mode would leave mail in SYSCOM and exit. MCIDAS then would notice the mail and call GRINIT to reallocate memory appropriately and call PINIT and SETP as described above.

Under the current implementation, however, the user is not permitted to dynamically modify the number of image/graphics frames within a PC-McIDAS session, so it is possible to divorce GRINIT from MCIDAS itself. GRINIT is now spawned as a separate program to reduce the size of MCIDAS.EXE.

Graphics point and video line drawing are implemented by accessing the EGA/VGA hardware directly. See the chapter "Using the IBM EGA and VGA".

Graphics line segment drawing is not implemented in PVEGA. See the discussion below of subroutine DRWLIN.

PLOTPACK -- The Plot Package for PC-McIDAS

The standard plot package entry points familiar to mainframe McIDAS programmers are implemented in PC-McIDAS (see PLOTPACK.FOR). For the most part, the entry points act as McIDAS programmers expect them to. A few comments should be made, however, about INITPL, PLOT, and ENDPLT.

INITPL on the EGA/VGA causes the frame automatically to be brought to the screen. It also erases the cursor automatically.

PLOT on the SSEC/Dataram tower calls GRLINE to send a packet to the 8085; the line segment is drawn by the 8085. PLOT on the EGA/VGA calls a Fortran subroutine called DRWLIN that generates the appropriate calls to P, the graphics point drawer. DRWLIN also takes care of omitting points outside the window. When the line segment being drawn is entirely within the window and has a one-pixel width with no dashing, DRWLIN uses Bresenham's Algorithm for maximum efficiency. DRWLIN is also one of the few places in PC-McIDAS where a large number of INTEGER*2 (not INTEGER*4) variables are used -- this is also done to increase efficiency. Some applications call DRWLIN directly, I believe, so this entry point should probably be retained.

In the EGA/VGA implementation, pixels are drawn in graphics memory only. They are not saved in the frame space in extended memory until ENDPLT is called. ENDPLT also takes care of re-drawing the cursor.

USING THE IBM EGA AND VGA

Among the imagery/graphics devices supported by PC-McIDAS are the IBM Enhanced Graphics Adapter (EGA) and the IBM Video Graphics Array (VGA). For both of these devices it is necessary, if we are to get adequate performance, to interact directly with the graphics hardware rather than use the BIOS graphics video functions. The purpose of this chapter is to describe how to program the EGA and VGA hardware.

At the time of this writing, PC-McIDAS uses the VGA as a glorified EGA. The VGA supports all the EGA's graphics modes, and PC-McIDAS uses only mode 16 (350 lines by 640 elements by 16 colors), an EGA mode. In most respects, the EGA and VGA are programmed identically in mode 16. The only differences arise in handling color palette selection.

The first few sections below will describe the programming considerations common to both the EGA and VGA in mode 16. Following them will be a section on EGA palette selection, then one on VGA palette selection.

Programming Considerations Common to Both the EGA and VGA

The following sections will assume graphics mode 16 and a fully-populated EGA/VGA memory, without bothering to say so repeatedly. Much of what will be said does not actually require these assumptions, but trying to provide a completely general exposition would be more trouble than it's worth.

Graphics Memory Organization

Graphics memory is organized in two pages (0-1). Page 0 begins at segment 0A000H; page 1 begins at segment 0A800H.

Within a page, memory is organized in 4 bit planes (0-3). That is, the bit 0's of all pixels in a page are stored contiguously, followed by all the bit 1's, etc. The purpose of the bit plane construct is to allow the various bit planes within a page to share the same address space, reducing by a factor of 4 the address space required. When a program accesses a particular address in graphics memory, it may access any of the 4 bit planes at that address. Which bit plane is actually accessed is determined by a register setting, as described below.

Within a given bit plane, each byte contains 1 bit from each of 8 pixels. Each byte is organized as follows: the high-order bit corresponds to the leftmost pixel, the low-order bit to the rightmost. Pixels are stored left-to-right across a screen row; screen rows are stored top-to-bottom.

Reading or writing to a bit plane is a two-stage process. First, execute an OUT instruction to a command register that specifies whether you want to read or write. Second, execute an OUT instruction to a so-called "map" register that defines which bit plane you want to access. After these two steps have been done, you can address the graphics memory just like any other part of memory.

Reading a Pixel

To read a pixel, you must read each of the 4 bytes (one per bit plane) containing the 4 bits making up the pixel value, extract from each of these bytes the bit corresponding to the pixel in question, and combine the pixel's bits to make a nibble containing the pixel value.

Only one step requires knowledge of the EGA/VGA hardware: reading a byte from a bit plane.

Suppose registers have been initialized as follows:

```
DS = segment address of page
SI = offset within bit plane of desired byte
        (i.e. ROW*80 + COL/8)
BL = bit plane (0-3)
```

To read the appropriate byte for the pixel, do the following:

```
MOV  DX,3CEH         ; port for command register to
                     ;   read bit planes
MOV  AL,4            ; command to enable reading of
                     ;   bit planes
OUT  DX,AL           ; enable bit plane reading

MOV  DX,3CFH         ; read map select register
MOV  AL,BL           ; number of bit plane (0-3)
OUT  DX,AL           ; select the bit plane for
                     ;   reading

MOV  AL,DS:[SI]      ; get the desired byte
```

Writing a Pixel

Writing a pixel is more than just the inverse of reading one. To write a pixel, you need to modify 1 bit in each of 4 bytes (1 byte for each bit plane). But in each of the 4 bytes, you want to modify only the bit corresponding to the pixel in question. The other bits must retain their old values.

The natural thing to try is to set up the graphics memory for a write operation, then use AND/OR instructions to set the appropriate bits directly. This does not work, however. When the graphics memory is set up for writing, it cannot be read. The AND/OR instructions do not work correctly; they return values as if the byte in question was clear to start with.

As a result, writing a pixel must be a read-modify-write operation. You must first read the current value of the byte you want to write to, then modify the appropriate bit, and finally write the modified byte back to the graphics memory.

Reading a byte is described in the previous section. Writing a byte is done similarly.

Suppose registers have been initialized as follows:

```
ES = segment address of page
DI = offset within bit plane of desired byte
        (i.e. ROW*80 + COL/8)
CL = bit plane (0-3)
CH = byte value to be written
```

To write the byte to the bit plane, do the following:

```
MOV  DX,3C4H          ; port for command register to
                      ;  write bit planes
MOV  AL,2             ; command to enable writing of
                      ;  bit planes
OUT  DX,AL            ; enable bit plane writing

MOV  DX,3C5H          ; map mask register
MOV  AL,1
SHL  AL,CL            ; need a 1 in the bit corres-
                      ;  ponding to the bit plane
OUT  DX,AL            ; select the bit plane for
                      ;  writing
```

```
MOV   ES:[DI],CH        ; store the value
```

NOTE: When reading a bit plane, you set the map register to the number of the bit plane itself. When writing a bit plane, you set the map register to have a 1 in the bit corresponding to the bit plane.

<u>Writing an Image</u>

When writing a single pixel, you are required to preserve the neighboring pixels. This necessitates the kind of read-modify-write implementation described above.

When writing an entire image, however, there is no such requirement, so you do not need to read bytes before writing them. You can set up a bit plane for writing, then move in a whole string of bytes at once using the MOVS or STOS instructions in block-move mode.

PC-McIDAS implements full-image writing in two different ways, depending on context. TV control writes images a whole bit plane at a time. Commands like DF and RSTI, however, write them a line at a time.

Writing an image a whole bit plane at a time is more efficient, since one sets the EGA/VGA registers only once per bit plane per image. Writing an image a line at a time requires setting the registers once per bit plane per each line of the image. However, the former method, despite its greater efficiency, cannot be used to load an image that is visible on the screen while it is being loaded. It causes the image colors to flash as the various bit planes are loaded, because the interval between successive refreshes of the screen is much less than the length of time required to load the image.

<u>Important PC-McIDAS Modules That Read/Write the EGA/VGA</u>

The driver that handles reading or writing a pixel is PVEGA.ASM.

A quarter-tone image is loaded via HLFTON.ASM. A 16-level image is loaded via HRSIMG.ASM.

Important routines involved in saving an image to a disk file are SAVHRS.ASM, BITPLN.ASM, and EXTMOV.ASM. For restoring a previously saved image see RSTHRS.ASM.

TVEGA.ASM, the TV control module for EGA/VGA workstations, also accesses the EGA/VGA hardware directly in a variety of ways.

Color Selection on the EGA

In graphics mode 16, the EGA permits 16 colors to be displayed at one time. These 16 colors can be selected freely from a set of 64 colors supported by the hardware. (Each color consists of 1 bit each for R, G, B and 1 bit each for R', G', B', the latter being intensity bits.)

A set of 16 selected colors is known as a "palette". Palette selection is accomplished via the BIOS video interrupt, using the subfunction defined by AH=10H, AL=2. 17 values can actually be defined; the 17th is the "overscan" color, which for PC-McIDAS is always 0, or black.

The palette for each PC-McIDAS frame is stored in the Frame Palette Block of SYSCOM. TVEGA takes care of setting the correct palette each time an image is brought to the screen. Note that different frames can employ different color palettes.

When an image is saved to a disk file using the SAVI command, the palette is saved with it, and the palette is restored when the image is restored using the RSTI command.

Color Selection on the VGA

The VGA supports a much greater number of possible colors than does the EGA: 64 * 64 * 64 = 262,144 colors rather than just 64.

The color selection process occurs in two stages on the VGA. You still specify a palette of 16 colors out of 64, but you can also specify the 64 available colors themselves by giving 6-bit red, green, and blue intensity levels (the so-called "color register" values) for each. The latter selection process occurs via the BIOS video interrupt, using the subfunction defined by AH=10H, AL=12H.

PC-McIDAS turns the two-stage color selection process back into a one-stage process. The palette is always set to colors 0-15. Color selection then amounts to setting the color registers for colors 0-15. (We do not care what colors are set for 16-63 since these are never used.) This means specifying 48 one-byte values -- the red, green, and blue intensity levels for each of 16 color levels.

When a VGA is being used, the Frame Palette Block in SYSCOM is given a different interpretation from that given when an EGA is used. In the EGA setting, there are 16 sets of 16 bytes -- one 16-color palette for each of 16 frames. In the VGA setting, there are 16 sets of 48 bytes -- 16 (R,G,B) triplets for each of 16 frames.

When a VGA image is saved to a disk file using the SAVI command, the 48 color register values are saved, and they are restored if the image is restored using the RSTI command.

THE KEYBOARD FILTER

PC-McIDAS Keyboard Requirements

There are two functional requirements that necessitate special handling of keyboard input in PC-McIDAS:

1) When a single-letter command is entered using the **Alt** key, the command must be executed immediately, even if other keystrokes precede it in the typeahead buffer. Similarly, if a key on the keypad is pressed to switch to a new text window, etc., the keystroke must not wait in line in the typeahead buffer.

2) **Ctrl-S** and **Ctrl-Q** must be implemented to stop and start text output to the screen. Since PC-McIDAS by-passes BIOS and DOS in writing text output to the text window interface, PC-McIDAS must handle Ctrl-S and Ctrl-Q on its own. Ctrl-S and Ctrl-Q must take effect immediately, even if other keystrokes precede them in the typeahead buffer.

Satisfying these functional requirements depends, in both cases, on the systems software being able to look ahead in the typeahead buffer and filter out keystrokes requiring immediate attention. The module that makes possible such a filtering operation is called KBIOSF.EXE.

Functional Description of KBIOSF

Programs, including DOS itself, ordinarily access keyboard input via BIOS INT 16H. The particular function performed by INT 16H depends on the value in the AH register, as follows:

AH=0 -- Return the next available keystroke

AH=1 -- Indicate if a keystroke is waiting

AH=2 -- Get the current shift status code

(The last function is not used by PC-McIDAS.)

The basic idea of KBIOSF is to replace the BIOS INT 16H with a new handler that provides the same functions as does INT 16H, but provides lookahead functions as well.

KBIOSF maintains two local typeahead buffers. The reason for two buffers is to allow two filtering passes: one to implement Ctrl-S/Ctrl-Q handling, the other to implement single-letter command handling and text window/soft tablet control. These two buffers will be referred to as the "1st pass buffer" and the "2nd pass buffer". More will be said about these buffers below.

KBIOSF implements the three functions implemented by BIOS INT 16H -- as it must do to provide the interface expected by DOS and other non PC-McIDAS programs. The only difference is that the functions for AH=0 and AH=1 look not in the BIOS typeahead buffer but in the 2nd pass buffer. The following additional functions are defined:

AH=80H -- Enable keyboard filter

AH=81H -- Disable keyboard filter

AH=82H -- Get the next available keystroke, if any, from the 1st pass buffer.

AH=83H -- Put a keystroke in the 2nd pass buffer.

AH=84H -- Get the next available keystroke, if any, from the BIOS typeahead buffer.

AH=85H -- Put a keystroke in the 1st pass buffer.

AH=86H -- Initialize

Each keystroke moves through the filter in the following way. The Ctrl-S/Ctrl-Q handler takes a key from the BIOS typeahead buffer via function AH=84H. If the key is Ctrl-S or Ctrl-Q it is handled and thrown away. If not, it is put into the 1st pass buffer via function AH=85H. The single-letter command handler takes the key from the 1st pass buffer via function AH=82H. If it is a key that can be handled in the background, it is handled and thrown away. Otherwise, it is put into the 2nd pass buffer via function AH=83H. This makes the key available to applications programs that access keystrokes through the normal function AH=0.

In other words, in the 1st pass buffer all Ctrl-S and Ctrl-Q characters have been filtered out. In the 2nd pass buffer, all single-letter commands handled by TVCTRL and all keypad keystrokes for controlling the text windows and soft tablet have also been filtered out.

Single-Letter Command Handling

On each tick, TVCTRL triggers KBIOSF to get a key from the 1st pass buffer.

If a key is returned, TVCTRL looks to see if it is an **Alt** key for a single-letter command handled by TVCTRL; if so, TVCTRL handles it and throws it away. TVCTRL also looks to see if the key is one of the keypad keys used to control the text window interface and soft data tablet. If so, TVCTRL triggers SCRINI (see the chapter on "The Text Window Interface") and throws the key away.

Otherwise, TVCTRL triggers KBIOSF to put the key into the 2nd pass buffer to make it available to applications programs and the scanner.

Ctrl-S and Ctrl-Q Handling

If Ctrl-S is pressed, text output to the screen is suspended until another keystroke is entered. If the latter keystroke is a Ctrl-Q or another Ctrl-S, it is thrown away. The implementation of this functionality is a bit complicated.

There is a flag in the Terminal Control Block of SYSCOM (the Ctrl-S flag) that indicates if a Ctrl-S is currently active. (I.e. the flag=1 if and only if text output is currently suspended by a Ctrl-S.) There are three places in the system where the state of this flag makes a difference:

1) in VIDEO.EXE, the BIOS INT 10H replacement,

2) in SCRINI.EXE, the text window interface handler,

3) in KBIOSF.EXE, the keyboard filter.

Moreover, there are three places in the system where the state of the flag can be changed:

1) in VIDEO.EXE, the BIOS INT 10H replacement,

2) in TVCTRL, and

3) in NXTKEY, the subroutine that applications call to get the next keystroke.

Each time VIDEO is triggered to write text to the screen, it takes the next waiting keystroke (if any) from the BIOS typeahead buffer and sets or clears the Ctrl-S flag, as appropriate, depending on what keystroke it gets and on the existing state of the Ctrl-S flag. If the keystroke is neither Ctrl-S nor Ctrl-Q, VIDEO triggers KBIOSF to put the keystroke into the 1st pass buffer.

Then, if the Ctrl-S flag is set, VIDEO goes into a loop, scanning the BIOS typeahead buffer, as above, until the Ctrl-S flag clears. By having this kind of Ctrl-S/Ctrl-Q handling in VIDEO itself, the system is assured of getting immediate response to Ctrl-S/Ctrl-Q keystrokes. Moreover, the system is not suspended by Ctrl-S unless some process actually attempts text output to the screen.

So, VIDEO takes care of Ctrl-S and Ctrl-Q handling so long as VIDEO is being called. Note that VIDEO also takes care of getting keys from BIOS and stuffing them into the 1st pass buffer. But what if no text output is currently being generated, so VIDEO is not being called? Clearly, some other process must also get keys from BIOS; otherwise, the keyboard would go dead whenever VIDEO is not being called.

The other process that scans the BIOS typeahead buffer is TVCTRL. It also checks for Ctrl-S and Ctrl-Q before putting a key into the first pass buffer. It is necessary to have TVCTRL, as well as VIDEO, check for Ctrl-S/Ctrl-Q. Otherwise, there is a race condition: if TVCTRL gets its hands a Ctrl-S before VIDEO does, the Ctrl-S is not noticed until control returns to the MCIDAS.EXE level and NXTKEY is called. TVCTRL's Ctrl-S/Ctrl-Q handling differs from VIDEO's in that TVCTRL does not loop if the Ctrl-S flag is set. There is no need to loop until some process actually triggers VIDEO to send some text to the screen.

SCRINI, the text window interface handler that is triggered on every "tick", does not bother to refresh the text screen if the Ctrl-S flag is set. The screen cannot change while the Ctrl-S flag is set -- a process that wanted to change the screen would have to call VIDEO and would loop there -- so refreshing the screen on every tick would be a waste of machine cycles.

Finally, there is a time-out associated with Ctrl-S. When the Ctrl-S flag is set, SCREENS increments a counter on each tick. If the counter times out, SCREENS clears the Ctrl-S flag. KBIOSF clears the counter on each keystroke.

THE PC-McIDAS COMMAND SCANNER

Overview

PC-McIDAS commands may be entered at the keyboard or through one of a variety of user interfaces. In either event, a "command line" is generated -- i.e. a sequence of characters representing a PC-McIDAS command or sequence of commands together with the command parameters. The PC-McIDAS system must be able to accept the command line, determine what action is appropriate to process the PC-McIDAS command or sequence of commands, and cause the appropriate action to be undertaken. The system routines responsible for handling command lines in this way are known collectively as "the scanner".

In PC-McIDAS, the scanner comprises a separate executable module, MCIDAS.EXE (see MCIDAS.FOR, SCSCNX.FOR, etc.). There are also some non-scanner functions included in MCIDAS.EXE, but these are unimportant for the present discussion. To run PC-McIDAS, one runs MCIDAS.EXE.

MCIDAS.EXE consists essentially of a big loop that looks for input from the keyboard or one of the user interfaces. When a full command line has been received, MCIDAS calls SCSCNX to parse the command line. The action taken depends on the kind of command found. It may be passed to TVCTRL or the mainframe, or an executable module may be activated locally on the workstation.

Input to the Scanner

There are various ways in which a command line may be received by the scanner.

1) It can be received a character at a time through a sequence of calls to GETKEY. The characters returned by GETKEY can arise in several ways:

a) They can be entered at the keyboard (returned by calls to NXTKEY).

b) They can be contained in an active batch file activated by the RUN command.

c) They can be obtained from the string table when a function key or a single-letter numeric key is pressed.

A sequence of characters returned by GETKEY becomes a command line when a carriage return is encountered.

2) A command line can be sent from a host computer and posted in SYSCOM by the communications software. Such a command is waiting if LOOKB(2,82).NE.0, in which case the scanner calls COMKYN to retrieve the command.

3) A command line can be generated by a user interface and posted in SYSCOM. Such a command is waiting if LOOKB(2,243).NE.0. In that event, the scanner calls USRKYN to retrieve the command. This same mechanism is also used by commands that want to leave mail to spawn another command when they complete.

4) A command line can be generated by a voice-recognition interface and posted in SYSCOM. Such a command is waiting if LOOKB(5,0).NE.0.

Parsing a Command Line

When the scanner has received a command line, it calls SCSCNX to parse the command line and take appropriate action. Other routines relevant for the parsing process are MCTOKN, MCTOK2, KYNANL, and TOKANL. String expansion is also done at this time.

SCSCNX loops through the sequence of PC-McIDAS commands contained in the command line. Each command is parsed and the command parameters are stored in SYSCOM. The command is then sent to the host (via SNDKYN) or executed locally on the workstation (via KSPAWN), as appropriate.

If any command in a sequence of commands leaves mail in SYSCOM to start another command, that new command is handled before continuing with the sequence of commands. This is necessary so the "mail" is not lost if a later command in the sequence also leaves mail to start a command.

SPAWNING SUBPROCESSES

Overview

There are a variety of scenarios in which one PC-McIDAS program (the parent process) needs to start up (spawn) another PC-McIDAS program (a child process, or subprocess). Three main kinds of spawn occur in PC-McIDAS. The principal mechanism used to implement each kind of spawn is the SPAWN utility in the MicroSoft C Library.

In the first kind of spawn, the parent process remains in memory and causes the child process to run as an ordinary DOS executable. The parent is suspended until the child completes, then the parent resumes. One example: all PC-McIDAS commands are spawned by the scanner. Another example: many PC-McIDAS commands spawn sub-commands transparently to the user.

When the first kind of spawn is used, the parent and child are both in memory at the same time. Consider, for example, the DFG command. When the DFG command line is entered, MCIDAS spawns DFG; then DFG spawns LODIMG, say. While LODIMG is running, all three programs -- MCIDAS, DFG, and LODIMG -- are in memory. It often happens in such cases that a parent does not have sufficient memory available to spawn a needed child.

The second kind of spawn handles such cases. Here, the parent does not remain in memory while the child runs. Instead, the parent leaves mail in SYSCOM and exits, and PC-McIDAS takes care of spawning the child. There is some loss of flexibility in that the child must run after the parent has completed. This has not been a very great hindrance in practice, however. An example of this kind of spawn is provided by commands like IGTV that leave mail to run MAP after they complete.

The last kind of spawn is involved in the PC-McIDAS implementation of the SQW facility of mainframe McIDAS. SQW allows a program to dynamically link in subroutines at run-time. PC-McIDAS implements SQW by spawning a child on the first call to SQW, then simply jumping to the child on subsequent calls. This mechanism is used to link dynamically to navigation modules.

The First Kind of Spawn -- Parent Stays in Memory

The underlying modules here are ISPAWN, KSPAWN, and LSPAWN. Each calls the MicroSoft C Library SPAWN utility to spawn a child process.

ISPAWN is called by ISQX. KSPAWN is called by SCSCNX, JSQX (indirectly), and KSQX (indirectly). LSPAWN is called by LSQX.

ISPAWN simply spawns the child process. If the child program is not found on the PC, it is **not** sent to the host.

KSPAWN does what ISPAWN does, and it also re-enables CTRL-BREAK checking for the child. CTRL-BREAK checking is disabled while MCIDAS is running to prevent the user from accidentally aborting MCIDAS itself. When MCIDAS spawns a PC-McIDAS command, it must use KSPAWN (which it does, indirectly, via SCSCNX) to enable the user to CTRL-BREAK out of the PC-McIDAS command, if needed. A child uses ISQX or ISPAWN, since CTRL-BREAK checking has already been enabled for the child.

LSPAWN is the same as ISPAWN except that if the child program is not found on the PC, and if the PC is configured to communicate back to a host computer, a packet will be sent to run the child on the host. LSPAWN is called by LSQX, which is called by DUO, for example.

When the child to be spawned is a DOS command, the SYSTEM entry point in the MicroSoft C Library is used, instead of SPAWN. SYSTEM loads a new copy of the DOS command processor COMMAND.COM. A PC-McIDAS entry DOSCMD is provided to enable PC-McIDAS commands easily to spawn DOS commands.

The Second Kind of Spawn -- Parent Leaves Mail in SYSCOM

Suppose a PC-McIDAS command CMD1 wants another command CMD2 to run immediately following completion of CMD1. Then CMD1 should include code like the following:

```
      CHARACTER CMD(160)
         ...
     (Put the text for CMD2, including parameters and
      trailing blanks, in the array CMD)
         ...
C---- Store CMD2 in SYSCOM
      CALL POKES(2,244,CMD,0,160)
C---- Set SYSCOM flag to indicate a command is pending
      CALL POKEB(2,243,1)
```

When CMD1 exits, CMD2 will be run. The various routines (ISQX, JSQX, KSQX, SCSCNX) that may have been used to start up CMD1 all include (directly or indirectly) code to check SYSCOM for a pending command and spawn it. Note that the command will be spawned from CMD1's parent, not necessarily from the PC-McIDAS scanner.

The Third Kind of Spawn -- Dynamically-Linked Subroutines

DOS does not directly support any form of dynamic linking of subroutines at run-time, so the implementation of SQW presents some problems.

The calling sequence of SQW is: CALL SQW(CPGM,N,M). N and M are arrays of arbitrary size through which parameters may be passed. The intent is that CPGM will be called as a subroutine with N and M passed as parameters. Moreover, if CPGM is SQW'ed repeatedly by a single command there should not be a lot of overhead associated with calls after the first one.

The basic idea of the PC-McIDAS implementation is to make CPGM a separate DOS executable that is spawned on the first call and jumped to on subsequent calls. Among other things, SQW stores pointers to N and M in SYSCOM so CPGM can find them.

Suppose you want to SQW a command named CPGM. Put the body of CPGM in a subroutine CPGM1, say, with the following calling sequence:

```
 SUBROUTINE CPGM1 (N,M)
 (The actual code for CPGM goes here.)
 END
```

CPGM itself will be a stub consisting of the following:

```
      PROGRAM CPGM
      EXTERNAL CPGM1
      DATA IFIRST/1/
      IF (IFIRST.EQ.1) THEN
         CALL PRGLOC
         IFIRST=0
      ENDIF
      CALL LNKSQW(CPGM1,LOOKDW(3,854),LOOKDW(3,858))
      GOTO 1000
      CALL CPGM1
 1000 CONTINUE
      END
```

LNKSQW is an assembler routine that sets up a call to CPGM1 using the addresses of N and M. Note that the CALL CPGM1 statement above label 1000 is never executed. It is a kludge to cause the linker to link in the subroutine CPGM1. (It is assumed here that CPGM1 resides in a separate Fortran module.) This is admittedly a gross cobble, but appears to be necessary.

PRGLOC is a routine that determines where the SQW'ed program is stored in memory and saves this information in SYSCOM. The first time SQW is called, the MicroSoft C Library SPAWN utility is used to load and execute the SQW'ed program.

However, subsequent calls to SQW with the same value for the CPGM parameter do not go thru SPAWN again. Instead, an assembler routine called PRGCAL gets the data that PRGLOC stored in SYSCOM, sets up registers as needed, and jumps to the SQW'ed program's location in memory. <u>**It is necessary, therefore, that no other SPAWNs have been done in the meantime.**</u> The great advantage of this scheme, of course, is that it enables one to use SQW to dynamically "link" in code (e.g. navigation transformations) that can then be invoked repeatedly without incurring the overhead of re-loading it each time it is called.

Calling Sequences of Spawn-Related Routines

SUBROUTINE **ABORT**(RETURN_CODE)
Abort a process with return code.

SUBROUTINE **ABRTCD**(ITYPE,ICODE)
Return the abort type and return code of a subprocess.

SUBROUTINE **DOSCMD**(COMMAND,STATUS)
Spawn a DOS command. COMMAND=CHARACTER(160).

FUNCTION **ISPAWN**(CPROGRAM_NAME)
Append .EXE to CPROGRAM_NAME and spawn it. Return status through function value.

FUNCTION **ISQX**(CPROGRAM,NUM_TOKENS_IN_CTOKEN_ARRAY, CTOKEN_ARRAY)
Spawn PC-McIDAS command named in CPROGRAM. CTOKEN is array of command line tokens.

FUNCTION **JSQX**(COMMAND)
Spawn PC-McIDAS command(s). COMMAND=CHARACTER(160).

FUNCTION **KSPAWN**(CPROGRAM_NAME)
Same as ISPAWN except re-enables CTRL-BREAK checking for child.

FUNCTION **KSQX**(COMMAND,COMMAND_LENGTH)
Same as JSQX except does not require 160 character input parameter. Use especially when passing a string constant for COMMAND.

FUNCTION **LNKSQW**(SUBROUTINE_NAME,NADDRESS,MADDRESS)
Explained in section on Dynamically-Linked Subroutines.

FUNCTION **LSPAWN**(CPROGRAM_NAME)
Same as ISPAWN except will send command to host if not found on workstation.

FUNCTION **LSQX**(CPROGRAM,NUM_TOKENS_IN_CTOKEN_ARRAY, CTOKEN_ARRAY)
Same as ISQX except will send command to host if not found on workstation.

FUNCTION **PRGCAL**(PSP_SEGMENT,DATA_SEGMENT)
Explained in section on Dynamically-Linked Subroutines.

FUNCTION **PRGLOC**
Explained in section on Dynamically-Linked Subroutines.

FUNCTION **SPWNER**(STATUS,CPROGRAM_NAME)
Handle errors encountered in spawning subprocesses.

FUNCTION **SQW**(CPROGRAM,N,M)
Spawns a subprocess allowing arbitrary arrays N, M to be passed. Explained in section on Dynamically-Linked Subroutines.

DOS FUNCTIONS

Introduction

Various PC-McIDAS applications and system modules need to have access to DOS functions -- e.g. to perform file I/O. Two main issues arise in connection with the use of DOS functions:

1) DOS functions as implemented in DOS 3.X are not re-entrant. I.e., DOS is not implemented in a way that allows it to interrupt itself, to have one process start up a DOS function when another process' DOS function is already in progress. However, PC-McIDAS needs to use DOS functions asynchronously in the background, so some means must be provided to serialize access to DOS functions.

2) It is desirable for applications programs to be ignorant of the details of the operating system. To the maximum extent possible, applications' source code should be operating system independent.

The following sections describe how PC-McIDAS has dealt with these two issues.

Serializing Access to DOS Functions

As described above, it is necessary to prevent an asynchronous background task from initiating a DOS function when the background task has interrupted a foreground task that is in the process of using a DOS function. The background task needs to wait its turn; i.e. access to DOS functions must be serialized.

To accomplish this serialization of access, a front-end, DOSFUNC.EXE, is installed under INT 21H, the general software interrupt vector for DOS functions. The front-end sets a semaphore and jumps to the code formerly installed under INT 21H. When control returns to the front-end, the semaphore is cleared. DOSFUNC must be careful to emulate correctly the INT and IRET instructions to make the front-end transparent to client processes (which may not even be PC-McIDAS programs). In particular, the flags register cannot be stepped on. The semaphore is stored in SYSCOM, so is accessible by all processes.

Any background task (TVCTRL, COMM, SCREENS) that needs to use a DOS function first checks the semaphore to see if another DOS function is in progress. If so, the background process waits until the next "tick" and tries again.

The most common case is that COMM has received a data packet that needs to be filed away on the hard disk, requiring the use of DOS functions. If the semaphore is set, COMM will hold the packet over until the next tick, at which time it will treat the packet as if it just came in. If necessary, the packet will be held over for a number of consecutive ticks.

Applications Interface to DOS Functions

PC-McIDAS includes various Fortran-callable subroutines that allow applications programs to access DOS functions without the applications' having to include DOS implementation details in their source code.

Actually, most of the DOS function interface subroutines are seldom, if ever, called directly by applications programs themselves. Rather, they are hidden below another interface layer. For example, most PC-McIDAS commands use the LW-file interface instead of directly calling routines like FOPEN, FREAD, etc. Similarly, no PC-McIDAS applications directly call routines like GETMEM and FREMEM.

The DOS function interface subroutines are important to the systems programmer, however. For example, they make it possible to convert the LW-file utilities to another operating system simply by replacing FOPEN, FREAD, etc.

The available interface subroutines are summarized below:

<u>File Directory Management</u>

SUBROUTINE **FATTRI** (CFLNAM,IATTRB,ISTAT) - Function 43H.
Set a file attribute.

SUBROUTINE **FEXIST** (CFLNAM,ISTAT) - Function 4EH.
Determine if the named file already exists.

SUBROUTINE **FFIRST** (CFLNAM,CENTRY,ISTAT) - Function 4EH.
Return first directory entry matching the file name.

SUBROUTINE **FNEXT** (CFLNAM,CENTRY,ISTAT) - Function 4FH.
Return next directory entry matching the file name.

SUBROUTINE **FRENAM** (CNAME1,CNAME2,ISTAT) - Function 56H.
Rename a file.

SUBROUTINE **FSIZE** (IHANDL,ISIZE) - Function 42H.
Return the size in bytes of a file.

SUBROUTINE **GATTRI** (CFLNAM,IATTRB,ISTAT) - Function 43H.
Return a file attribute.

SUBROUTINE **GETDSK** (AVAIL,CLUSTR,BYTES,SECTOR) - Function 36H. Return info about free space on disk drive.

INTEGER FUNCTION **IFRDSK** - Function 36H.
Return number of bytes available on default disk drive.

File Creation

SUBROUTINE **FCLOSE** (IHANDL,ISTAT) - Function 3EH.
Close a file handle.

SUBROUTINE **FCREAT** (CFLNAM,ISTAT,IHANDL) - Function 3CH.
Create a file handle.

SUBROUTINE **FDELET** (CFLNAM,ISTAT) - Function 41H.
Delete a file handle.

SUBROUTINE **FOPEN** (CFLNAM,ISTAT,IHANDL) - Function 3DH.
Open a file handle.

SUBROUTINE **FTEMP** (CPATH,ISTAT) - Function 5AH.
Create a uniquely named temporary file.

File I/O

SUBROUTINE **EPOINT** (IHANDL,ISTAT) - Function 42H.
Move file pointer to end-of-file.

SUBROUTINE **FPOINT** (IHANDL,OFFSET,ISTAT) - Function 42H.
Move file pointer to a given offset.

SUBROUTINE **FREAD** (IHANDL,NBYTES,IBUFF,ISTAT,NUMRD) - Function 3FH. Read bytes from a file.

SUBROUTINE **FWRITE** (IHANDL,NBYTES,IBUFF,ISTAT,NUMWR) - Function 40H. Write bytes to a file.

Flow of control

SUBROUTINE **ABORT** (ICODE) - Function 4CH.
Abort a program, passing ICODE as the return code.

SUBROUTINE **ABRTCD** (ITYPE,ICODE) - Function 4DH.
Return the abort type and code of a subprocess.

Memory Management

SUBROUTINE **FREMEM** (IADDR,ISTAT) - Function 49H.
Free a block of memory previously allocated via GETMEM.

SUBROUTINE **GETMEM** (NBYTES,IADDR) - Function 48H.
Request memory allocation.

SUBROUTINE **MEMAMT** (PARAS,ISTAT) - Function 48H.
Return number of paragraphs in largest block of memory currently available.

Time and Date

SUBROUTINE **GETTIM** (HHMMSS) - Function 2CH.
Return current time.

SUBROUTINE **GETYMD** (YYMMDD) - Function 2AH.
Return current date.

Interrupt Vectors

SUBROUTINE **GETVCT** (INTNUM,SEGMNT,OFFSET) - Function 35H. Return an interrupt vector.

SUBROUTINE **SETVCT** (INTNUM,SEGMNT,OFFSET) - Function 25H. Set an interrupt vector.

Miscellaneous

SUBROUTINE **CTRBRK** (IFLAG) - Function 33H.
Enable or disable ctrl-break checking.

SUBROUTINE **DOSPRM** (CPARMS,LENGTH) - Function 62H.
Return the parameters from the DOS command line.

SUBROUTINE **GTPRMS** (CPARMS,NPARMS) - Function 62H.
Return the first NPARMS parameters from the DOS command line.

SUBROUTINE **GETENV** (BUFFER,NBYTES) - Function 62H.
Return the first NBYTES bytes of the DOS Environment.

SUBROUTINE **GETSEG** (PSPSEG,DATSEG) - Function 62H.
Return PSP address and data segment address for currently executing process.

SUBROUTINE **SETDTA** - Function 1AH.
Set Disk Transfer Address to its default location.

THE PC-McIDAS UTILITY LAYER

Introduction

There are a variety of utility subroutines available to mainframe McIDAS programs. These utilities are known collectively as the "utility layer". To facilitate the porting of mainframe McIDAS source code to the PC-McIDAS environment, it was necessary first of all to recreate the utility layer in PC-McIDAS.

Most of the utilities are intrinsically dependent on the operating system and/or the hardware architecture (e.g., byte addressing within words) and therefore had to be rewritten for PC-McIDAS. Every effort was made to reproduce faithfully the calling sequences and functionality of the various routines.

In some cases, however, calling sequences had to be modified. MicroSoft Fortran does not, at the time of this writing, support variable length character strings and the various functions related to such strings. In particular, when a string is passed to a subroutine, the called subroutine has no way to determine the string's length. Either the caller and callee must agree always to pass a string of a particular length, or the calling sequence must include an argument that specifies the string length.

For certain utility routines (e.g. INDEX), therefore, it was necessary to modify the calling sequence to add a length argument. In all such cases, the name of the utility was changed (e.g. INDEX became JINDEX). Had the names been left unchanged, there would have been no automatic way to detect instances where a programmer porting a mainframe McIDAS module neglected to modify a calling sequence appropriately. The linker does not check in any way that calling sequences agree across modules. Nor is any run-time error message generated. By changing the names, we create a situation in which unmodified calls will give rise to "Undefined Reference" errors at link-time. It is strongly recommended that this practice be continued.

Various utility routines, grouped by function, are described below. Not included are utilities ordinarily used only by specialized subsystems of PC-McIDAS -- e.g. MCTOKN, which is called by the scanner, or SKIO, which is called by the scheduler.

Also not included here are specialized utilities for interfacing with DOS functions -- e.g. FREAD, FWRITE, etc. See the chapter "DOS Functions". Similarly, BIOS interface utilities -- e.g. SETMOD, SETPAL -- are described elsewhere. See the chapter "BIOS Functions".

In what follows, variable names are chosen to be descriptive, not necessarily to follow Fortran name-length or implicit typing conventions.

Assume the following statements are in effect:

```
IMPLICIT INTEGER (A-B,D-Z)
IMPLICIT CHARACTER*12 (C)
```

Variables with names like COLOR or COLUMN are integers, however, and variables named CHAR are CHARACTER*1. Assume all integer variables are 4-byte integers, unless otherwise indicated.

SYSCOM Access

INTEGER FUNCTION **LOOKB**(BLOCK,OFFSET)
Retrieve a 1-byte, unsigned SYSCOM value.

INTEGER FUNCTION **LOOKDW**(BLOCK,OFFSET)
Retrieve a 4-byte SYSCOM value.

SUBROUTINE **LOOKS**(BLOCK,OFFSET,DESTINATION_ARRAY, STARTING_OFFSET_WITHIN_ARRAY,NUM_BYTES)
Retrieve an arbitrary number of bytes from SYSCOM.

INTEGER FUNCTION **LOOKSB**(BLOCK,OFFSET)
Retrieve a 1-byte, signed SYSCOM value (i.e. sign-extend).

INTEGER FUNCTION **LOOKSW**(BLOCK,OFFSET)
Retrieve a 2-byte, signed SYSCOM value (i.e. sign-extend)

INTEGER FUNCTION **LOOKW**(BLOCK,OFFSET)
Retrieve a 2-byte, unsigned SYSCOM value.

SUBROUTINE **POKEB**(BLOCK,OFFSET,VALUE)
Store a 1-byte SYSCOM value.

SUBROUTINE **POKEDW**(BLOCK,OFFSET,VALUE)
Store a 4-byte SYSCOM value.

SUBROUTINE **POKES**(BLOCK,OFFSET,SOURCE_ARRAY,
  STARTING_OFFSET_WITHIN_ARRAY,NUM_BYTES)
  Store an arbitrary number of bytes in SYSCOM.

SUBROUTINE **POKEW**(BLOCK,OFFSET,VALUE)
  Store a 2-byte SYSCOM value.

McIDAS Command Parameter Retrieval

FUNCTION **CKWP**(CKEYWORD,ARGUMENT_NUM,CDEFAULT)
  Return character string keyword parameter.

FUNCTION **CPP**(ARGUMENT_NUM,CDEFAULT)
  Return character string positional parameter.

SUBROUTINE **CQFLD**(CSTRING)
  Return quote field. CSTRING=CHARACTER(160)

REAL*8 FUNCTION **DKWPHR**(CKEYWORD,ARGUMENT_NUM,DDEFAULT)
  Return double-precision real positional time
  parameter. DDEFAULT=REAL*8.

REAL*8 FUNCTION **DPP**(ARGUMENT_NUM,DDEFAULT)
  Return double-precision real positional parameter.
  DDEFAULT=REAL*8.

REAL*8 FUNCTION **DPPHR**(ARGUMENT_NUM,DDEFAULT)
  Return double-precision real positional time
  parameter. DDEFAULT=REAL*8.

REAL*8 FUNCTION **DPPLL**(ARGUMENT_NUM,DDEFAULT)
  Return double-precision real positional lat/lon
  parameter. DDEFAULT=REAL*8.

FUNCTION **IKWP**(CKEYWORD,ARGUMENT_NUM,IDEFAULT)
  Return integer positional parameter.

SUBROUTINE **INIKYN**
  Must be called at beginning of any PC-McIDAS command.
  Parameter-passing will not work without it.

FUNCTION **IPP**(ARGUMENT_NUM,IDEFAULT)
  Return integer positional parameter.

FUNCTION **IPPYD**(ARGUMENT_NUM,IDEFAULT)
  Return integer positional date parameter.

SUBROUTINE **KWNAMS**(DIMENSION_OF_CARRAY,NUM_KEYWORDS_FOUND, CARRAY)
Return names of all keywords in command line, except DEV=.

FUNCTION **NKWP**(CKEYWORD)
Return number of values associated with a keyword.

SUBROUTINE **UNPARS**(COMMAND)
Reconstruct command text from parameters in SYSCOM. COMMAND=CHARACTER(160)

LW File System

FUNCTION **LBI**(CFILENAME,BEGIN_BYTE,NUM_BYTES,IARRAY)
Read bytes from an LW-file.

FUNCTION **LBO**(CFILENAME,BEGIN_BYTE,NUM_BYTES,IARRAY)
Write bytes to an LW-file.

FUNCTION **LWC**(CFILENAME)
Create an LW-file.

FUNCTION **LWCLOS**(CFILENAME)
Close an LW-file.

FUNCTION **LWD**(CFILENAME)
Delete an LW-file.

FUNCTION **LWEXIS**(CFILENAME)
Determine if a specified LW-file exists.

FUNCTION **LWI**(CFILENAME,BEGIN_WORD,NUM_WORDS,IARRAY)
Read 4-byte words from an LW-file.

FUNCTION **LWNAME**(CFILENAME)
Check an LW-file name for validity.

FUNCTION **LWO**(CFILENAME,BEGIN_WORD,NUM_WORDS,IARRAY)
Write 4-byte words to an LW-file.

FUNCTION **LWOPEN**(CFILENAME,IHANDLE)
Open an LW-file. Return the file handle.

FUNCTION **LWRNAM**(CFILENAME1,CFILENAME2)
Rename an LW-file.

Path Names

CHARACTER*80 FUNCTION **KYPATH**(CFILENAME)
Return ASCII file name with path prefix
C:\MCIDAS\COMMANDS.

CHARACTER*80 FUNCTION **KYPATZ**(CFILENAME)
Return ASCIZ file name with path prefix
C:\MCIDAS\COMMANDS.

CHARACTER*80 FUNCTION **LWPATH**(CFILENAME)
Return ASCII file name with path prefix
C:\MCIDAS\DATA.

CHARACTER*80 FUNCTION **LWPATZ**(CFILENAME)
Return ASCIZ file name with path prefix
C:\MCIDAS\DATA.

CHARACTER*12 FUNCTION **NOPATH**(CFILENAME_WITH_PATH)
Strip path prefix from ASCII file name.
CFILENAME_WITH_PATH=CHARACTER*80

CHARACTER*12 FUNCTION **NOPATZ**(CFILENAME_WITH_PATH)
Strip path prefix from ASCIZ file name.
CFILENAME_WITH_PATH=CHARACTER*80

CHARACTER*80 FUNCTION **STPATH**(CFILENAME)
Return ASCII file name with path prefix
C:\MCIDAS\SETUP.

CHARACTER*80 FUNCTION **STPATZ**(CFILENAME)
Return ASCIZ file name with path prefix
C:\MCIDAS\SETUP.

CHARACTER*80 FUNCTION **VTPATZ**(CFILENAME)
Return ASCIZ file name with path prefix
for root directory of RAM Disk.

Text Output

SUBROUTINE **CDEST**(CLINE,IVALUE,WINDOW,ROW,COLUMN,COLOR)
Display a line of text terminated by '$$'.
CLINE=CHARACTER(*)

SUBROUTINE **DDEST**(CLINE,IVALUE)
   Display a line of text terminated by '$$'.
   CLINE=CHARACTER(*)

SUBROUTINE **EDEST**(CLINE,IVALUE)
   Display a line of text terminated by '$$'.
   CLINE=CHARACTER(*)

SUBROUTINE **EDESTC**(CLINE,IVALUE,WINDOW,COLOR)
   Display a line of text terminated by '$$'.
   CLINE=CHARACTER(*)

SUBROUTINE **LTQ**(CLINE)
   Display or print a line of text terminated by '$$'.
   CLINE=CHARACTER(*)

SUBROUTINE **MSTQ**(CLINE,TEXT_ATTRIBUTE)
   Display a line of text terminated by '$$'.
   Replaces mainframe McIDAS TQ.
   CLINE=CHARACTER(*)

SUBROUTINE **PRINT**(CLINE)
   Print a line of text terminated by '$$'.
   CLINE=CHARACTER(*)

SUBROUTINE **SCRACK**(PAIR,CHAR,COLOR,BACKGROUND,BLINK)
   Decode a (character,attribute) pair.

SUBROUTINE **SDEST**(CLINE,IVALUE)
   Display a line of text terminated by '$$'.
   CLINE=CHARACTER(*)

SUBROUTINE **SDESTC**(CLINE,IVALUE,WINDOW,COLOR)
   Display a line of text terminated by '$$'.
   CLINE=CHARACTER(*)

SUBROUTINE **SDEST0**(CLINE,IVALUE)
   Display a line of text terminated by '$$'.
   Forces text to Window 0, color=white.
   CLINE=CHARACTER(*)

INTEGER*2 FUNCTION **SPACK**(CHAR,COLOR,BACKGROUND,BLINK)
   Encode a (character,attribute) pair.

FUNCTION **TQSET**(DEVICE)
   Set/examine current default display device.

SUBROUTINE **WNDINT**(AH_REGISTER,AL_REGISTER)
Invokes interrupt for text window interface.
E.g. CALL WNDINT(99,N) brings text window N to the screen. See SCREENS.ASM for complete list of functions.

Formatting Numerical Output

FUNCTION **CFD**(DVALUE,DECIMAL_PLACES)
Convert REAL*8 DVALUE to CHARACTER*12.

FUNCTION **CFE**(RVALUE,DECIMAL_PLACES)
Convert REAL*4 RVALUE to CHARACTER*12.

FUNCTION **CFF**(RVALUE,DECIMAL_PLACES)
Convert REAL*4 RVALUE to CHARACTER*12.

FUNCTION **CFI**(IVALUE)
Convert INTEGER IVALUE to CHARACTER*12, right-justified, leading blanks.

FUNCTION **CFU**(IVALUE)
Convert unknown IVALUE to CHARACTER*12, left-justified.

FUNCTION **CFZ**(IVALUE)
Convert INTEGER IVALUE to hexadecimal CHARACTER*12, right-justified, four leading blanks.

FUNCTION **CLFI**(IVALUE)
Convert INTEGER IVALUE to CHARACTER*12, left-justified.

SUBROUTINE **CLZERO**(CTEXT,LENGTH)
Replace leading blanks with text 0's.
CTEXT=CHARACTER(*)

FUNCTION **NDIGS**(IVALUE)
Returns number of digits in text representation of INTEGER IVALUE.

Date and Time

SUBROUTINE **CONVRT**(IVALUE,CTEXT)
Convert HHMMSS time or DDDMMSS lat/lon value to HH:MM or DDD:MM format, left-justified.

SUBROUTINE **GETDAY**(YYDDD)
Return current Julian date.

SUBROUTINE **GETTIM**(HHMMSS)
Return current time.

SUBROUTINE **GETYMD**(YYMMDD)
Return current date.

FUNCTION **IDMYYD**(IDAY,IMONTH,IYEAR)
Return Julian date.

Variable Type Conversion Routines

REAL*4 FUNCTION **ALIT**(C)
Convert CHARACTER*4 to bitwise identical REAL*4.

FUNCTION **BCD**(I)
Convert integer to binary coded decimal.

CHARACTER*4 FUNCTION **CLIT**(I)
Convert integer/real to bitwise identical CHARACTER*4.

REAL*8 FUNCTION **DFTOK**(C)
Convert CHARACTER*12 text representation of a numerical token to REAL*8.

REAL*8 FUNCTION **DLIT**(C)
Convert CHARACTER*8 to bitwise identical REAL*8.

FUNCTION **IDROND**(D)
Round a REAL*8 value.

FUNCTION **IFTOK**(C)
Convert CHARACTER*12 text representation of a numerical token to INTEGER*4 (rounded).

SUBROUTINE **II**(FIELD_WIDTH,IVALUE,CSTRING,
OFFSET_IN_CSTRING)
Convert integer to text string.

FUNCTION **IROUND**(X)
Round a REAL*4 value.

FUNCTION **LIT**(C)
Convert CHARACTER*4 to bitwise identical INTEGER*4.

Basic Byte-Move Routines

SUBROUTINE **BIGMOV**(SOURCE_SEGMENT,DEST_SEGMENT,NUM_BYTES)
Move bytes using segment addresses. NUM_BYTES is allowed to exceed 64K.

SUBROUTINE **EXTMOV**(SOURCE_ADDR,DEST_ADDR,NUM_WORDS)
Move 2-byte words. ADDR's are 24-bit physical addresses, including addresses in extended memory.

SUBROUTINE **MOVB**(NUM_BYTES,SOURCE,DEST,DEST_OFFSET)
Move bytes.

SUBROUTINE **MOVC**(NUM_BYTES,SOURCE,SOURCE_OFFSET,DEST, DEST_OFFSET)
Move bytes.

SUBROUTINE **MOVCR**(NUM_BYTES,SOURCE,SOURCE_OFFSET,DEST, DEST_OFFSET)
Same as MOVC, but move is made right-to-left. Use when SOURCE is to left of DEST and they overlap.

SUBROUTINE **MOVPTR**(NUM_BYTES,SOURCE_ADDR,DEST_ADDR)
Move bytes, using pointers to source and destination.

SUBROUTINE **MOVW**(NUM_WORDS,SOURCE,DEST)
Move 4-byte words.

SUBROUTINE **MOVWR**(NUM_WORDS,SOURCE,DEST)
Same as MOVW, but move is made right-to-left. Use when SOURCE is to left of DEST and they overlap.

SUBROUTINE **MVARSG**(SOURCE,DEST_SEGMENT,NUM_BYTES)
Move bytes from array to segment.

SUBROUTINE **MVPAD**(NUM_BYTES,SOURCE,SOURCE_OFFSET,DEST, DEST_OFFSET,DEST_LENGTH)
Move bytes, padding to end of DEST with blanks.

SUBROUTINE **MVPADR**(NUM_BYTES,SOURCE,SOURCE_OFFSET,DEST,
DEST_OFFSET,DEST_LENGTH)
Same as MVPAD, but move is made right-to-left. Use when SOURCE is to left of DEST and they overlap.

SUBROUTINE **MVSGAR**(SOURCE_SEGMENT,DEST,NUM_BYTES)
Move bytes from segment to array.

SUBROUTINE **MVSGSG**(SOURCE_SEGMENT,DEST_SEGMENT,NUM_BYTES)
Move bytes from segment to segment.

Pack and Crack Routines

SUBROUTINE **CRACK**(NUM_BYTES,SOURCE,DEST)
Crack byte array into INTEGER*4 array.

SUBROUTINE **CRACK2**(NUM_BYTES,SOURCE,DEST)
Crack byte array into INTEGER*2 array.

SUBROUTINE **CRACKB**(NUM_ITEMS,SOURCE,SOURCE_OFFSET_IN_BITS,
NUM_BITS_PER_ITEM,DEST,SIZE_OF_DEST_VALUES_IN_BYTES,
SIGN_EXTEND_FLAG)
Crack bits into a BYTE, INTEGER*2, or INTEGER*4 array. It is assumed that source bits are stored in the order natural for IBM 4381.

SUBROUTINE **CRACKN**(NUM_ITEMS,HI_LO_FLAG,SOURCE,DEST,
SIZE_OF_DEST_VALUES_IN_BYTES)
Crack nibbles into BYTE, INTEGER*2, or INTEGER*4 array. HI_LO_FLAG=1 means crack most significant nibble first.

SUBROUTINE **PACK**(NUM_BYTES,SOURCE,DEST)
Pack least significant byte of INTEGER*4 array values into a byte array.

SUBROUTINE **PACK2**(NUM_BYTES,SOURCE,DEST)
Pack least significant byte of INTEGER*2 array values into a byte array.

SUBROUTINE **SWBYT2**(IARRAY,NUM_WORDS)
Reverses the order of bytes in each 2-byte word of IARRAY.

SUBROUTINE **SWBYT4**(IARRAY,NUM_WORDS)
Reverses the order of bytes in each 4-byte word of IARRAY.

Logical AND, OR, etc.

SUBROUTINE **FLAND**(IARG1,IARG2)
Return in IARG1 the logical AND of arguments.

SUBROUTINE **FLOR**(IARG1,IARG2)
Return in IARG1 the logical OR of arguments.

SUBROUTINE **FLXOR**(IARG1,IARG2)
Return in IARG1 the logical XOR of arguments.

FUNCTION **LAND**(IARG1,IARG2)
Return logical AND of arguments.

FUNCTION **LOR**(IARG1,IARG2)
Return logical OR of arguments.

FUNCTION **LXOR**(IARG1,IARG2)
Return logical XOR of arguments.

Other Byte and Character Manipulation Routines

SUBROUTINE **BLKA**(NUM_4_BYTE_WORDS,IARRAY)
Fill with ASCII blanks.

SUBROUTINE **CLEANA**(NUM_BYTES,IARRAY)
Change unprintable characters to blanks.

SUBROUTINE **CLEANW**(NUM_4_BYTE_WORDS,IARRAY)
Change unprintable characters to blanks.

SUBROUTINE **ERASE**(SEGMENT,NUM_BYTES)
Zero out a section of memory.

FUNCTION **IC**(CSTRING,OFFSET)
Extract a character from a string.

FUNCTION **ISAN**(IARG)
Returns 1 if and only if all 4 bytes of IARG are ASCII alphanumeric (A-Z,0-9,blank).

FUNCTION **ISBLNK**(CSTRING,LENGTH_IN_BYTES)
Returns 1 if and only if CSTRING consists entirely of ASCII blanks.

FUNCTION **ISCANS**(CSTRING,LENGTH_IN_BYTES,CHAR)
Returns 1-based offset of first occurrence of CHAR in CSTRING. 0 if not found.

FUNCTION **ISCHAR**(IARG)
Returns 1 if and only if all 4 bytes of IARG are printable ASCII characters.

FUNCTION **JCMPS**(NUM_BYTES,STRING1,STRING1_OFFSET,STRING2, STRING2_OFFSET)
Returns 1 if and only if STRING1 and STRING2 are identical through the specified number of bytes.

FUNCTION **JINDEX**(CSTRING1,LENGTH1,CSTRING2,LENGTH2)
Returns 1-based offset of first occurrence of CSTRING2 in CSTRING1. 0 if not found. Replaces mainframe's **INDEX** function.

FUNCTION **NMCHAR**(CSTRING,LENGTH_OF_CSTRING, OFFSET_OF_FIRST_CHAR,OFFSET_OF_LAST_CHAR)
Return the number of characters in a string, plus offsets of first and last char. Replaces mainframe's **NCHARS** function.

SUBROUTINE **SQUEEZ**(CTEXT,LENGTH)
Compresses text by compressing strings of consecutive blanks to a single blank. Modifies LENGTH accordingly.

SUBROUTINE **STC**(IVALUE,CSTRING,OFFSET)
Store least significant byte of IVALUE at 0-based OFFSET in CSTRING.

SUBROUTINE **STOREC**(REPEAT_COUNT,BYTE,DEST,DEST_OFFSET)
Store consecutive copies of BYTE (e.g. to fill with 0's or blanks).

SUBROUTINE **UPCASE**(CHAR)
Convert CHAR to upper case.

SUBROUTINE **ZEROW**(NUM_4_BYTE_WORDS,IARRAY)
Zero out an array.

Keyboard

SUBROUTINE **CAPLOF**
Turn CAPS LOCK off.

SUBROUTINE **CAPLON**
Turn CAPS LOCK on.

SUBROUTINE **CLTYBF**
Clear typeahead buffer.

SUBROUTINE **GETKEY**(ISCAN,IASCII)
Get next keystroke from keyboard, batch file, or function key string, as appropriate.

SUBROUTINE **GETKBD**(ISCAN,IASCII)
Get next keystroke from keyboard.

SUBROUTINE **NUMLOF**
Turn NUM LOCK off.

SUBROUTINE **NUMLON**
Turn NUM LOCK on.

SUBROUTINE **NXTKEY**(ISCAN,IASCII)
Get next keystroke from keyboard; handle CTRL-S and CTRL-Q.

Communications

SUBROUTINE **RCVTXT**(BUFFER,PACKET_LENGTH,STATUS)
Return a packet from async comm. BUFFER should be at least 768 bytes. STATUS: 0=success, 1=data lost, 80h=no packet available.

SUBROUTINE **QRQRQR**(BUFFER)
Send F0-protocol packet(s) to host.

SUBROUTINE **SENOUT**(BUFFER)
Send 02/03-protocol packet(s) to tower. Buffer terminated by ETX (=03).

SUBROUTINE **SERIAL**(CSTRING,IVALUE)
Send string terminated by '$' and text representation of IVALUE to serial port 1. For debugging.

SUBROUTINE **SNDKYN**(COMMAND,STATUS)
Send McIDAS command(s) back to host in TRB-packet.

SUBROUTINE **SNDMSG**(MESSAGE,NODE_NUMBER,STATUS)
Send message to another node. Status < 0 == failed.

SUBROUTINE **SNDTXT**(COMMAND,STATUS)
Send McIDAS command(s) back to host as pure text.

SUBROUTINE **SNDXOF**
Send an XOFF.

SUBROUTINE **SNDXON**
Send an XON.

Graphics

SUBROUTINE **DRWLIN**(DEVICE_NUMBER,FRAME,BEG_LINE,BEG_ELEM,
END_LINE,END_ELEM,COLOR,WIDTH,IPLOT,DASH_FLAG,
INIT_FLAG)
Draw a line segment.

SUBROUTINE **ENDPLT**
Same as mainframe ENDPLT.

SUBROUTINE **ERASEG**(FRAME)
Erase a frame.

SUBROUTINE **GRLINE**(FRAME,COLOR,BEG_LINE,BEG_ELEM,
END_LINE,END_ELEM,WIDTH,DASH_LENGTH,GAP_LENGTH,
GAP_COLOR)
Draw a line segment on tower-based workstation.
Should not be called by applications, since it is
device-dependent. (Called by DRWLIN).

SUBROUTINE **INITPL**(FRAME,WIDTH)
Same as mainframe INITPL.

SUBROUTINE **PLOT**(LINE,ELEM,PEN)
Same as mainframe PLOT.

SUBROUTINE **WRTEXT**(UPPER_LEFT_LINE,UPPER_LEFT_ELEM,
HEIGHT,CTEXT,NUM_CHARS,COLOR)
Draw text on graphics.

Saving and Restoring Images and Graphics

SUBROUTINE **GETPIC**(CFILENAME,FRAME)
Append .PIC extension to filename. If file exists,
restore its image to frame.

SUBROUTINE **RSTIMG**(FRAME,CFILENAME)
Restore a saved image.

SUBROUTINE **SAVIMG**(FRAME,CFILENAME)
Save an image.

SUBROUTINE **SAVPIC**(CFILENAME,FRAME)
Append .PIC extension to filename; save image in file.

Spawning Subprocesses

SUBROUTINE **ABORT**(RETURN_CODE)
Abort a process with return code.

SUBROUTINE **DOSCMD**(COMMAND,STATUS)
Spawn a DOS command. COMMAND=CHARACTER(160).

FUNCTION **ISPAWN**(CPROGRAM_NAME)
Append .EXE to CPROGRAM_NAME and spawn it. Return status through function value.

FUNCTION **ISQX**(CPROGRAM,NUM_TOKENS_IN_CTOKEN_ARRAY, CTOKEN_ARRAY)
Spawn PC-McIDAS command named in CPROGRAM. CTOKEN is array of command line tokens.

FUNCTION **JSQX**(COMMAND)
Spawn PC-McIDAS command(s). COMMAND=CHARACTER(160).

FUNCTION **KSPAWN**(CPROGRAM_NAME)
Same as ISPAWN except re-enables CTRL-BREAK checking for child.

FUNCTION **KSQX**(COMMAND,COMMAND_LENGTH)
Same as JSQX except does not require 160 character input parameter. Use especially when passing a string constant for COMMAND.

FUNCTION **LNKSQW**(SUBROUTINE_NAME,NADDRESS,MADDRESS)
Explained in chapter "Spawning Subprocesses".

FUNCTION **LSPAWN**(CPROGRAM_NAME)
Same as ISPAWN except will send command to host if not found on workstation.

FUNCTION **LSQX**(CPROGRAM,NUM_TOKENS_IN_CTOKEN_ARRAY,
    CTOKEN_ARRAY)
    Same as ISQX except will send command to host if not found on workstation.

FUNCTION **PRGCAL**(PSP_SEGMENT,DATA_SEGMENT)
    Explained in chapter "Spawning Subprocesses".

FUNCTION **PRGLOC**
    Explained in chapter "Spawning Subprocesses".

FUNCTION **SPWNER**(STATUS,CPROGRAM_NAME)
    Handle errors encountered in spawning subprocesses.

FUNCTION **SQW**(CPROGRAM,N,M)
    Spawns a subprocess allowing arbitrary arrays N, M to be passed. Explained in chapter "Spawning Subprocesses".

Logging Events

SUBROUTINE **STAMP**(CTEXT)
    Insert date/time stamp at beginning of CTEXT. CTEXT=CHARACTER*80.

SUBROUTINE **UNILOG**(CTEXT)
    Add message to UNIDATA.LOG file. CTEXT=CHARACTER*80.

Frame Control

SUBROUTINE **DSPFRM**
    Force current frame to be refreshed. Used on EGA/VEGA workstations; for example, when palette is changed.

FUNCTION **OPPFRM**
    Returns number of frame opposite to current frame.

SUBROUTINE **SETFRM**(FRAME)
    Set current frame to the given frame number.

SUBROUTINE **SHOFRM**(FRAME)
    Force frame to screen.

Lock and Unlock

SUBROUTINE **LOCK**(CNAME)
Stub for compatibility with mainframe.

SUBROUTINE **LOCKR**(CNAME)
Stub for compatibility with mainframe.

SUBROUTINE **SLOCK**(CNAME)
Stub for compatibility with mainframe.

SUBROUTINE **UNLOCK**(CNAME)
Stub for compatibility with mainframe.

Sound Production

SUBROUTINE **BEEP**
Produce a beep sound.

SUBROUTINE **SOUND**(FREQUENCY,TWENTIETHS_OF_A_SECOND)
Produce a tone.

Device Status Checks

SUBROUTINE **CHKFLP**(IBUFF,ISTAT)
Return status of floppy drive. IBUFF=INTEGER(1000).
ISTAT=128 if drive not ready.

FUNCTION **PRSTAT**()
Returns printer status.

Addressing Utilities

FUNCTION **LOCVAR**(VARIABLE)
Returns segment:offset address of VARIABLE.

FUNCTION **PHYSAD**(LOCVAR)
Returns 24-bit physical address corresponding to real-mode segment:offset address in LOCVAR.

FUNCTION **SEGVAR**(VARIABLE)
Returns segment address of VARIABLE.

Timing Control

SUBROUTINE **DELAY**(TWENTIETHS_OF_A_SECOND)
Delay using BIOS INT 15H. This routine is suspect -- BIOS INT 15H is apparently non-reentrant. Since COMM uses INT 15H also, DELAY shuts down communications while it's waiting. For an alternative, see **WAIT**, below.

SUBROUTINE **WAIT**(TWENTIETHS_OF_A_SECOND)
Wait for specified period of time. This routine uses TV control ticks rather than BIOS INT 15H (see **DELAY**, above). Actually, CALL WAIT(1) waits 0 sec. to .05 sec., CALL WAIT(2) waits .05 sec. to .1 sec., etc. Hence, one should probably use a minimum of 2 for the input value.

Miscellaneous

SUBROUTINE **BRKPNT**
Trip the DEBUG breakpoint. For debugging.

SUBROUTINE **SETCLK**(BCD_CENTURY,BCD_YEAR,BCD_MONTH,BCD_DAY, BCD_HOURS,BCD_MINUTES,BCD_SECONDS,YEAR,MONTH,DAY, HOURS,MINUTES,SECONDS)
Set both CMOS clock and DOS date/time.
See FUNCTION **BCD**(IVALUE).

SUBROUTINE **STDERR**(ERROR_STATUS)
Produce error messages for standard DOS error codes.

SUBROUTINE **TRMNL**(ITERM)
Return terminal number.

SUBROUTINE **USRMOU**
Polls mouse, setting user mouse values in SYSCOM.

EGA/VGA GRAPHICS AND IMAGERY

Introduction

EGA/VGA-based PC-McIDAS workstations are capable of generating images and graphics locally and of receiving and displaying images and graphics generated on a host computer.

Local graphics (as opposed to imagery) generation is discussed in the chapter "Graphics Drivers". Various details concerning the EGA/VGA hardware are discussed in the chapter "Using the IBM EGA and VGA".

The purpose of this chapter is to describe, for EGA/VGA-based PC-McIDAS workstations: how images (as opposed to graphics) are generated locally; how host-generated images and graphics are handled; and how images and graphics are saved to hard disk and later restored.

Generating Images Locally

To display an image locally, one could call the graphics driver for every point or every line in the image. This turns out to be too slow, however.

For optimal performance, two assembly language routines were created for displaying images. One, called HLFTON, handles "quarter-toned" images; the other, called HRSIMG, handles images displayed directly. HLFTON is called by a PC-McIDAS command LODHFT, which in turn is spawned by DFG (spawned by XXDF). Similarly, HRSIMG is called by a PC-McIDAS command LODIMG, which is also spawned by DFG.

Each of these routines writes directly to the graphics memory. Each is capable of displaying up to 64K of data per call. Each passes the data through a look-up table, making it possible to enhance the image without modifying the underlying data.

The calling sequences are as follows:

```
CALL HRSIMG(MODE,PAGE,IMAGE_DATA,LEVELS,LINES,ELEMS,
        SCRLIN,SCRELE)

  where:
    MODE == graphics mode...13, 14, 15, or 16
            (PC-McIDAS only uses mode 16)
    PAGE == page in graphics memory (0 or 1)
    IMAGE_DATA == array of image data, 8 bits per
            pixel
```

```
          LEVELS == look-up table (see below)
          LINES == number of image lines in IMAGE_DATA
                    to display on this call (<= 64K of
                    data per call)
          ELEMS == number of elements per line
          SCRLIN == starting line on screen (0-based)
          SCRELE == starting elem within starting line

     CALL HLFTON(PAGE,IMAGE_DATA,LEVELS,LINES,ELEMS,SCRLIN)
          where:
               PAGE == page in graphics memory (0 or 1)
               IMAGE_DATA == array of image data, 8 bits per
                    pixel
               LEVELS == look-up table (see below)
               LINES == number of image lines in IMAGE_DATA
                    to display on this call ( <= 64K of
                    data per call)
               ELEMS == number of elements per line
               SCRLIN == starting line on screen (0-based)
```

Everything here is self-explanatory except for LEVELS, the look-up table. LEVELS is an array of 256 bytes.

In the case of HRSIMG, LEVELS maps 8-bit data values to graphics levels 0-15. A program can set the LEVELS any way it wants, but the usual case is to set it up to give a linear mapping based on a specified minimum level, maximum level, and number of levels. There is a Fortran-callable subroutine SETLVL (no parameters) that sets up a LEVELS array for this usual case. It receives its parameters via a COMMON block:

```
     COMMON /PALCOM/LEVELS(256),PALETS(17),CUTOFF(13),
                    MINLVL,MAXLVL,NUMLVL
```

The relevant values here are MINLVL, MAXLVL, and NUMLVL. MINLVL and MAXLVL are 0-based. SETLVL stores values in the LEVELS array.

In the case of HLFTON, LEVELS maps 8-bit data values to the range 0-24, since quarter-toned images are capable of representing 25 apparent shades of grey (8 shades mixing black and dark grey, 8 mixing dark grey and light grey, 8 mixing light grey and white, plus 1 for all white).

For examples of the use of HRSIMG, HLFTON, and SETLVL, see the source files LODIMG.FOR and LODHFT.FOR.

Host-Generated Images and Graphics

EGA/VGA-based PC-McIDAS workstations that have a communications link to a host computer may receive images and graphics packets generated on the host. Such packets conform to the so-called F0-protocol. The following imagery and graphics routing codes are defined:

Routing code 30H -- Erase frame
  Byte 0 == Frame number

Routing code 31H -- Graphics line segment(s)
  Byte 0 == Frame number
  Byte 1 == Color
  Byte 2 == Dash length (0=solid)
  Byte 3 == Gap color (0=solid)
  Byte 4 == Gap length
  Byte 5 == Line width in pixels
  Bytes 6-7 == Number of line segments defined in this packet
  Bytes 8-9 == Starting line number (0-based) for segment 1
  Bytes 10-11 == Starting element number (0-based) for segment 1
  Bytes 12-13 == Ending line number (0-based) for segment 1
  Bytes 14-15 == Ending element number (0-based) for segment 1
  Bytes 16-17 == Ending line number (0-based) for segment 2
  Bytes 18-19 == Ending element number (0-based) for segment 2
  etc. (Line segments 2-N are assumed to each start where the previous segment ended.)

Routing code 32H -- Line of image data
  Byte 0 == Frame number
  Bytes 1-2 == Line number
  Byte 3 == Image type
  Bytes 5-N == Image data, 4 or 8 bits per pixel

  Image types: the various image types differ in whether they call HLFTON (quarter-toned images) or HRSIMG; the look-up table used; the color palette used; and the number of bits per pixel in the image data.

    Type 0 -- HRSIMG/14 levels/VIS.PAL/8 bits
    Type 1 -- HLFTON/25 levels/QUARTER.PAL/8 bits
    Type 2 -- HRSIMG/14 levels/IR.PAL/8 bits

Type 3 -- HRSIMG/256 levels/VIS.PAL/4 bits
Type 4 -- HRSIMG/256 levels/QUARTER.PAL/
4 bits
Type 5 -- HRSIMG/256 levels/IR.PAL/4 bits
Type 6 -- HRSIMG/16 levels/QUARTER.PAL/8 bits

(The "levels" item here is the NUMLVL value used in calling SETLVL with MINLVL=0, MAXLVL=255. See above.)

One possible approach to handling these packets would be to have the communications software display them immediately as they come in. There are various difficulties in this approach, however. For one thing, it would mean linking into the comm software a lot of image/graphics-handling code. Since the comm software is resident, this would entail a permanent loss of RAM even for users who never generate images/graphics on the host. For another thing, it would mean potentially interrupting a foreground task that is already writing to the screen on one frame and having the comm software attempt to write to another frame. This is potentially a very messy proposition.

The approach that was taken instead was to have the comm software store incoming image/graphics packets in a system LW-file **GQUEUE.SYS**. This graphics queue file is organized as a circular queue, with the head and tail pointers stored in SYSCOM, hence available to all tasks. Two entry points, GETPCK and PUTPCK, are used to retrieve and store packets in this file. (GETPCK and PUTPCK are also used in connection with the command queue file; see the chapter "The Command Queue".)

The scanner, in between PC-McIDAS commands, checks the graphics queue head and tail pointers in SYSCOM to see if there are packets waiting in the queue to be processed. If so, the scanner spawns a PC-McIDAS command **GPCKTS**. GPCKTS bails the graphics queue, displaying as it goes. GPCKTS takes control of the display, forcing the relevant frame to the screen and preventing (via a flag in byte 15 of block 2 of SYSCOM) the user from switching to another frame or window. Note that more packets may be coming in while GPCKTS is running; in fact, this is the usual case. The comm software will continue to file packets away in the background.

When GPCKTS finds it has emptied the queue completely, it delays for a short interval (the length of which depends on comm mode and baud rate) and retries before giving up and exitting. This is done to keep GPCKTS from having to be loaded repeatedly when there are short pauses in the comm stream.

GQUEUE.SYS is opened at initialization time and remains open throughout the PC-McIDAS session. The file handle for GQUEUE.SYS is stored in SYSCOM.

Saving and Restoring Images and Graphics

Images and graphics that are displayed on an EGA or VGA may be saved to a disk file for later recall. These so-called "picture" files store the image/graphic in bit plane format, so their recall to the screen is optimized.

Each picture file begins with a 128-byte header:

Byte 0 == Graphics mode
Bytes 1-48 == Color palette
Byte 49 == Type code

For EGA's, only 16 bytes are used for the color palette; the remaining 32 are undefined. In particular, the overscan register is neither saved nor restored. For VGA's, the palette values are 16 (R,G,B) triplets.

The type code = 0 if the picture is a graphic, 1 if the picture is an image. Images have 256 bytes of navigation data appended following the bit plane data.

Bit plane data is stored with bit plane 0 first, bit plane 3 last.

There are two subroutines, written in assembler for optimal performance, that handle the core of the saving and restoring process. They are SAVHRS and RSTHRS. They are called in turn by SAVIMG and RSTIMG, with the following calling sequences:

CALL **SAVIMG**(FRAME,LW_FILE_NAME)

CALL **RSTIMG**(FRAME,LW_FILE_NAME)

It is these latter routines that should be called by any PC-McIDAS command that needs to save/restore picture files.

RSTIMG will not permit a picture to be restored if the current graphics mode is different from the mode under which the picture was saved. It does not, however, defend against attempts to restore on a VGA a picture saved on an EGA, or **vice versa**.

Note that there exists a program SHOPIC.EXE that lets a user display a picture without having PC-McIDAS running or even installed. This lets a user who has EGA/VGA Shift-PrintScreen software get hardcopy of an image or graphic.

THE COMMAND QUEUE

Why is a Command Queue Needed?

PC-McIDAS workstations that have a communications link to a host computer may receive from the host computer requests to execute locally certain PC-McIDAS commands. Such host-generated commands may be received at the workstation more quickly than they can be serviced. There is no upper bound on the size of the backlog that could be generated. Some mechanism is needed so that host-generated commands do not fall on the floor if they cannot be executed right away.

One possible strategy -- in fact the stategy originally adopted by PC-McIDAS -- is for the communications software not to accept a host-generated command packet until it can be serviced. This is nice and simple, but unfortunately it can lead to a deadlock in ProNET-based workstations.

Suppose, for example, the host sends an LB command followed immediately by a CS command, and the CS is refused while the LB is running. The LB command, like certain other PC-McIDAS commands, needs to send a packet to the host. This is done to inform the host of the workstation's new loop bounds. Here's where the deadlock arises. The ProNET comm software is strictly half-duplex. Because it is refusing the packet for the CS command, the comm software remains in its receive state, and it cannot get out of that state until it accepts the CS command packet, which it cannot do until the LB command completes. The LB command cannot complete, however, until the comm software gets into its transmit state and sends to the host the packet generated by LB. Deadlock.

Command Queue Implementation

The solution adopted by PC-McIDAS is to accept all host-generated commands as they come in. They are queued up in a system LW-file called **QUEUE.SYS**. QUEUE.SYS is a circular queue, and the head and tail pointers are stored in SYSCOM.

The communications software takes care of queueing and de-queueing packets. So long as the queue is non-empty, **all** incoming packets (other than image/graphics packets, which have their own queue) are stored in the command queue. This ensures that packets are handled in the order in which they were sent. There is one exception: IDLE packets are not queued at all, since they are no-ops.

Also, so long as the queue is non-empty, the comm software gives precedence to the packets in the queue. It eats packets from the head of the queue, and stores packets at the tail of the queue until it catches up and the queue is again empty.

There are two entry points, PUTPCK and GETPCK, for storing packets in and retrieving them from the queue file. These same entry points are also used for the graphics queue file, GQUEUE.SYS. QUEUE.SYS is opened at initialization time and remains open throughout the PC-McIDAS session.

Note that there is still a very small possibility, which will probably never be realized in practice, that the queue file, which is 16K bytes long, may fill up. What happens then? So long as the queue file is full, incoming packets are allowed to fall on the floor. At least that way there is no deadlock.

Note that it is essential that image/graphics packets are filed in their own queue, not the command queue. Otherwise, the command queue really might fill up, since image/graphics commands come thick and fast when they come. (See the chapter "EGA/VGA Graphics and Imagery" for a discussion of the queue for image/graphics packets.) What happens when the graphics queue gets full? In this case, incoming image/graphics packets are refused, not thrown away. This does not lead to a deadlock in practice since GPCKTS will be running, locking out commands like LB that might produce a deadlock.

ACCESSING EXTENDED MEMORY

Uses of Extended Memory in PC-McIDAS

PC-McIDAS runs in real-mode under DOS 3.X, so programs cannot execute in extended memory (i.e. memory above 1 megabyte in the address space). Extended memory can be used, however, to store data and programs, either by using a virtual (RAM) disk or by using BIOS function 15H, subfunction 87H. PC-McIDAS uses extended memory in the following ways:

1-2 megabytes -- RAM Disk containing the following DOS files:

> INTERF.EXE == drop-down menu HELP
>
> INTERFAC.DAT == data file for INTERF.EXE
>
> (Note that INTERF.EXE does not execute in extended memory. Its executable is stored there to enable it to be loaded more quickly than if it were stored on the hard disk. MCIDAS.EXE knows to load INTERF.EXE from the RAM disk, and INTERF.EXE knows to read its data from the RAM disk. I.e. these are "hard-wired".)

2 megabytes and up -- Various PC-McIDAS data structures accessed via BIOS INT 15H:

> 10 Text windows
>
> 10 Soft tablet windows
>
> 16 EGA/VGA frames (if applicable)
>
> (Note that these are not DOS files, nor is DOS even aware that this space has been "reserved". PC-McIDAS takes care of initializing this space (see SCRNEW.FOR) and maintaining its contents. It is "reserved" only in the sense that there are no other processes around to step on it.)

A natural question: Why aren't the data structures that are stored starting at 2 megabytes set up as DOS files using a RAM disk? The problem with that approach is that DOS is not re-entrant, so background processes would not be able to get access to those data structures when a foreground process was using a DOS function. There are a number of functional requirements of PC-McIDAS, however, that imply that background processes must be able to access the data structures in question any time they need to. These functional requirements include such things as being able to switch text windows or loop frames while foreground tasks are running.

EXTMOV -- How it Works and How to Use It

The files stored in RAM disk are accessed via ordinary DOS file I/O routines. The data structures stored at 2 megabytes and up, however, are accessed via BIOS INT 15H, subfunction 87H.

This BIOS function is described in the AT Technical Reference. It requires the caller to set up a block move Global Descriptor Table. The source and destination addresses are specified in 24-bit physical address form. The amount of data to move is specified in 2-byte words. Note that the BIOS function is capable of moving at most 64K bytes (32K words).

There is a PC-McIDAS subroutine EXTMOV that sets up the BIOS INT 15H call. EXTMOV has the following calling sequence:

CALL **EXTMOV**(SOURCE_ADDRESS,DEST_ADDRESS,NUM_WORDS)

The source and destination address are 24-bit physical addresses. To determine the 24-bit address of a variable or array in real-mode address space, use the following functions:

FUNCTION **LOCVAR**(VARIABLE)
Returns segment:offset address of VARIABLE.

FUNCTION **PHYSAD**(LOCVAR)
Returns 24-bit physical address corresponding to the real mode segment:offset address in LOCVAR.

For example, the phsical address of IARRAY, say, is given by:

ADDRESS=PHYSAD(LOCVAR(IARRAY))

EXTMOV sets up the call to the BIOS function. There is a granularity constant in EXTMOV that determines the amount of data moved per call to the BIOS. Like the BIOS function, EXTMOV handles at most 64K bytes per call, but it moves the data in chunks, according to the granularity. Interrupts are disabled by the BIOS function; the granularity makes it possible to tune EXTMOV if interrupts are being lost.

The BIOS function disables interrupts because it puts the 80286 into protected mode. Switching back to real mode is a slow process, so interrupts remain disabled for quite a long interval. No matter how the granularity is set, serial data is lost if the baud rate is high. To prevent the loss of serial data, EXTMOV sends an XOFF at the beginning and an XON at the end.

INITIALIZATION AND CONFIGURATION CONTROL

Workstation Configuration and the CONFIG Program

PC-McIDAS is designed to support a whole family of workstations with a variety of hardware/software configurations. A single, unified set of PC-McIDAS installation software is used for all workstations.

Some means must be provided to let a user specify the particular hardware configuration to be used in a given workstation. This is done via the program CONFIG.EXE in the \MCIDAS\SETUP subdirectory.

CONFIG steps the user through a series of questions about the workstation configuration. The responses are stored in the file \MCIDAS\SETUP\CONFIG.DAT. Each time CONFIG runs, it uses the existing version of CONFIG.DAT to supply the default responses, and it modifies CONFIG.DAT as needed as the user's responses change. Note that this means that if CONFIG.DAT gets lost or damaged, running CONFIG generally will not fix it. It is necessary, in such an instance, to copy CONFIG.DAT anew from the first installation diskette.

A user can run CONFIG as often as he/she pleases, e.g. to switch a workstation back and forth between a ProNET and an async comm link.

The CONFIG.DAT file is read by MCIDAS.EXE at run-time to initialize certain values in SYSCOM.

Besides modifying CONFIG.DAT, CONFIG does the following:

- constructs MCAUTO.BAT, the boot-time initialization batch file (see below)

- if the computer is an AT, copies AUTOEXEC.AT and CONFIG.AT from \MCIDAS\SETUP to AUTOEXEC.BAT and CONFIG.SYS, respectively, in the root directory

- if the computer is a PS/2, copies AUTOEXEC.PS2 and CONFIG.PS2 from \MCIDAS\SETUP to AUTOEXEC.BAT and CONFIG.SYS, respectively, in the root directory.

Note that \AUTOEXEC.BAT and \CONFIG.SYS get over-written each time CONFIG is executed. If a user wants to modify AUTOEXEC.BAT or CONFIG.SYS, therefore, the best way to do it is to modify the corresponding file back in \MCIDAS\SETUP.

MCAUTO.BAT -- Boot-Time Initialization of PC-McIDAS

At boot-time, the AUTOEXEC.BAT batch file invokes the batch file \MCAUTO.BAT. MCAUTO installs the PC-McIDAS device drivers, initializes the printer port, copies several data files, and starts up PC-McIDAS.

The MCAUTO.BAT is created by CONFIG.EXE. Its contents depend on the hardware/software configuration of the work-station. The contents are as follows:

ECHO OFF
CD C:\MCIDAS\SETUP
CHKENV
(CHKENV inspects the DOS Environment to see if it contains a string MCIDAS=INSTALLED. If so it aborts with errorlevel=1; otherwise errorlevel=0.)
IF ERRORLEVEL 1 GOTO INSTALL
ECHO PC-McIDAS device drivers have already been installed.
GOTO RUN
:INSTALL
NMLOFF
(NMLOFF turns NUM LOCK off.)
SYSCOM
(Installs SYSCOM. SYSCOM **must** be installed before other drivers.)
VIDEO
(Installs BIOS INT 10H replacement.)
KBIOSF
(Installs keyboard filter.)
DOSFUNC
(Installs DOS function semaphore front-end.)
SCRINI
(Installs SCREENS, the text window interface handler.)
ASYNC2
(Installs low-level async comm driver for port 2. If port 1 is to be used, this would be replaced by **ASYNC1**. If ProNET comm is to be used, this would be replaced by **PNETINT**. If no comm link is to be used, this would be deleted altogether.)
COMMA
(Installs high-level async comm driver. If ProNET comm is to be used, this would be replaced by **COMMP**. If no comm link is to be used, this would be replaced by **COMMN**.)
TVEGA
(Installs TV control for the EGA/VGA. If a tower were being used, this would be replaced by **TVSSEC**.)

PVEGA
(Installs graphics driver for the EGA/VGA. If a tower were being used, this would be replaced by **PVSSEC.**)

MODE COM1:96,N,8,1,P
(Initializes serial port 1 for printer. Actual command varies depending on port used, baud rate, etc.)

COPY INTERFAC.DAT E:
(Copies to the RAM disk the data file for the drop-down menu HELP. The letter used to designate the RAM disk varies depending on the number of other drives present.)

COPY INTERF.EXE E:
(Copies to the RAM disk the executable image for the drop-down menu HELP. The letter used to designate the RAM disk varies depending on the number of other drives present.)

COPY C:\MCIDAS\SETUP\UNIDATA.MNU C:\MCIDAS\SETUP\MENU.DAT
(Copies the menu file to be used. The name of the source file may vary. If the menu system is not to be used, this command will not appear.)

CHKINI
(Determines if the file \MCIDAS\SETUP\INITSYS.DAT exists. If so, it aborts with errorlevel=0; if not, errorlevel=1. INITSYS.DAT is deleted at boot-time to force a full workstation initialization on the first LOGON. CHKINI is used simply to avoid a disconcerting "File Not Found" error message from DOS when the file deletion is done.)

IF ERRORLEVEL 1 GOTO NODEL
DEL INITSYS.DAT
:NODEL
SET MCIDAS=INSTALLED
(Enters the string MCIDAS=INSTALLED into the DOS Environment, so that later invocations of MCAUTO before a reboot will not re-install device drivers.)

:RUN
CD C:\MCIDAS\COMMANDS
COMMAND /C MCIDAS
(Start up MCIDAS.EXE. COMMAND.COM is re-invoked to avoid certain DOS actions related to batch files. E.g., if a user Ctrl-Break's out of a PC-McIDAS command, we don't want DOS to butt in and ask if we want to abort the batch file (MCAUTO).)

Users should be discouraged from modifying MCAUTO.BAT. The reason MCAUTO.BAT was split off from AUTOEXEC.BAT, in fact, was to make it easy for users who need to modify AUTOEXEC.BAT to do so, with lessened likelihood of their disrupting the PC-McIDAS initialization process.

Run-Time Initialization of PC-McIDAS

When MCIDAS.EXE is invoked, it goes through a number of initialization steps. For some steps, it calls subroutines; for others, it spawns independent programs. The latter method is used where possible, because it helps reduce the size of MCIDAS.EXE. MCIDAS.EXE is resident and running throughout a PC-McIDAS session, so memory used by MCIDAS.EXE is memory that is unavailable for PC-McIDAS commands.

The main steps in run-time initialization are the following:

1) Check DOS Environment for string MCIDAS=INSTALLED to verify that device drivers have been installed.

2) Clear screen; display "Please stand by..."

3) Disable Ctrl-Break checking. (User is prevented from Ctrl-Break'ing out of MCIDAS.EXE; Ctrl-Break is re-enabled within PC-McIDAS commands, however.)

4) Install INT 16H interrupt vector for keyboard filter.

5) Zero out SYSCOM. (Note that this means SYSCOM values cannot be initialized at boot-time without some provision here to save-and-restore.)

6) Spawn MCINIT.EXE. See below.

7) Delete temporary files from \MCIDAS\DATA.

8) If using async comm, initialize comm here and send an XOFF. (XOFF sent so we don't lose data while initializing extended memory.)

9) Call SCINIT to initialize extended memory used by text window interface, etc. See below.

9) Spawn GRINIT.EXE to initialize graphics drivers. See below.

10) Start up TV control.

11) Clear the type-ahead buffer.

12) If using ProNET comm, reset and get on the ring.

13) Determine amount of available memory and display message.

14) If configured to transmit commands to a host, send transparent LOGOFF command.

15) If a menu-driven workstation, generate LOGON and MENU commands.

16) Enable text window interface.

17) If async, send an XON.

18) Enable high-level comm driver.

There are many other, minor initialization steps, consisting mostly of calls to POKEB, POKEW, or POKEDW to initialize various flags in SYSCOM. These are commented in the source code (see MCIDAS.FOR), and should be self-explanatory.

MCINIT.EXE, INITSYS.DAT, and CONFIG.DAT

Two data files are initimately connected with PC-McIDAS run-time initialization. Both are in subdirectory \MCIDAS\SETUP.

The first file is CONFIG.DAT. As has been discussed above, CONFIG.DAT contains a description of the hardware/software workstation configuration selected by the user. Its contents are modified via CONFIG.EXE.

The second file is INITSYS.DAT. When a user exits PC-McIDAS via the **EXIT** command, the contents of the Terminal Control Block (TCB) and Looping Control Block (LCB) of SYSCOM are saved in INITSYS.DAT. If MCIDAS.EXE is re-invoked, the TCB and LCB are restored, so loop bounds, etc., retain the values they had.

MCINIT calls three subroutines: SYSINI, LBINIT, and KYNLST. The most substantial of these is SYSINI. It is SYSINI that reads INITSYS.DAT, if INITSYS.DAT exists, and stores its data in SYSCOM. Having done so, SYSINI reads CONFIG.DAT and stores its data in SYSCOM. Note that this means that CONFIG.DAT takes precedence. SYSINI does a number of other SYSCOM initialization steps, such as initializing the palettes for EGA/VGA frames.

If no file INITSYS.DAT exists, SYSINI applies certain default values. In addition, it sets a flag that causes the first LOGON command to run through its full initialization sequence. Note that MCAUTO.BAT deletes INITSYS.DAT at boot-time. This forces a full initialization after each boot.

The second subroutine called by MCINIT is LBINIT. This initializes the internal copies of string table data.

The third subroutine called by MCINIT is KYNLST. This constructs the SYSCOM list of PC-McIDAS commands present in \MCIDAS\COMMANDS. This list is used by the scanner to determine if a given command should be spawned locally or sent to the host. As an aside, it may be mentioned that the **DOS** command in PC-McIDAS also calls KYNLST before exitting. This is done in case the DOS command has been used to add/delete a PC-McIDAS command to/from \MCIDAS\COMMANDS.

SCINIT and SCRNEW.EXE

MCINIT calls SCINIT to initialize the text window interface as well as the soft tablets and EGA/VGA frames. SCINIT spawns SCRNEW.EXE to do the extended memory initialization. Then SCINIT enables:

- VIDEO, the BIOS INT 10H replacement
- SCREENS, the text window interface handler
- INTERF, the drop-down menu HELP

SCINIT also calls DSPREL to display the initial PC-McIDAS text containing the release number. Note that this means that to change the release number one must modify DSPREL and relink MCINIT.EXE (not MCIDAS.EXE).

GRINIT.EXE

GRINIT initializes various values used by the PV graphics driver. See the source code.

The main point to be made here is that GRINIT has gone through a lengthy evolution, as a result of which its structure is somewhat obscure. At one time, EGA workstations stored their frames in real-mode RAM. The number of frames allocated could be changed dynamically during a PC-McIDAS session, via a command called **EGA**. To implement this, it was necessary to have EGA set a flag which was detected by MCIDAS.EXE when control returned from EGA. MCIDAS called GRINIT as a subroutine to actually deallocate the old frames and allocate and initialize the new ones. This had to be done from code linked into MCIDAS so that the allocated memory would belong to MCIDAS and therefore would not disappear with the completion of the EGA command.

There is still a lot of code in GRINIT to implement this earlier architecture. Since GRINIT is now spawned as a separate program this code will not work. That's the bad news. The good news is that this particular code will never be invoked any longer and since GRINIT is a separate, transient program the wasted code space is harmless. The code has been left in simply to save it for possible future use.

The LOGON Command and TRMINI

Logging on to a PC-McIDAS workstation can be a two-stage process. The LOGON command logs the user on to the workstation itself. Then, if the workstation is configured to transmit commands to a host, LOGON generates a LOGON command for the host computer and sends it off. The LOGON command sent to the host has parameters appended to it that identify the workstation type and software release level.

In addition, LOGON performs a number of initialization steps both locally and on the host computer. These initialization steps are handled chiefly by a subroutine called TRMINI.

For workstations configured to transmit commands to a host, TRMINI generates a number of host commands that are appended (separated by semicolons) to the LOGON sent to the host. Among these are such commands as:

- GD ... to set graphics defaults, especially the type of graphics device (SSEC or EGA)

- PCCLOC ... to cause the host to send back a PCCLOC command with the host's date and time appended; this synchronizes the workstation's clock with the host's

- LB ... to inform the host of the workstation's loop bounds

- ECHO ... to cause the host to send back the text message "Initialization completed." when the host commands have completed.

It is important that the commands are all sent in one command line (TRB) rather than in a series of command lines. Otherwise, there is no way to predict the order in which the commands will be executed on the host. Commands that execute before the host LOGON completes will be rejected.

DEBUGGING TOOLS

Using DEBUG.COM With PC-McIDAS

DEBUG.COM, the DOS debugger, is useful for debugging PC-McIDAS commands. Suppose you want to debug a command called BLAH.EXE within PC-McIDAS. Enter the following PC-McIDAS command:

```
DOS "DEBUG BLAH.EXE
```

This will invoke DEBUG. PC-McIDAS provides an entry point

```
SUBROUTINE BRKPNT
```

that can be inserted into the BLAH source code to set a breakpoint at a desired point. You cannot specify an address to BRKPNT; the breakpoint is simply tripped when BRKPNT is called. The breakpoint is an INT 3 instruction (HEX CC). When the breakpoint is triggered, use DEBUG's 'E' command to replace the INT 3 with a NOP (HEX 90) and step past it with the 'T' command. If desired, you can then use 'E' again to restore the NOP to an INT 3 if you expect the breakpoint to be hit again later and you want it to be tripped.

To trace through a Fortran program from the beginning, without a breakpoint, it helps to know a few things. Invoke DEBUG via

```
DEBUG BLAH.EXE
```

Then, enter the 'R' command. DEBUG will display the register contents. When DEBUG first loads a Fortran program, it initializes the segment registers to point to the Program Segment Prefix. To find the first Fortran instruction, you must add 100H to the PSP segment address. Recall, however, that the segment register contents are shifted by one hex digit. Hence, you add not 100H but 10H to the contents of DS. Suppose, for example, that DS contains the value 23A2H when BLAH.EXE is first loaded. The first Fortran instruction, then, is located at address 23B2:1. Enter the DEBUG command

```
G 23B2:1
```

to break at the first instruction. You can use T, P, and G to trace through from there.

Interpreting the assembler code generated by Fortran takes a little practice. The most easily recognized statements are subroutine and function calls. Calls are preceded by a series of PUSH'es -- a segment and offset (one PUSH each) are pushed for each parameter passed to the subroutine/function. It is easy, therefore, to see how many parameters a call has. This can be used to help you figure out where you are in the source code. To see what value is being passed as a parameter, a good place to trap is at the point at which the parameter address (address, note, not value) is being pushed on the stack. Function calls can be distinguished from subroutine calls, since immediately upon return from a function the AX and DX register contents are moved to memory (since the function value is returned through the AX and DX registers).

Another type of statement easily recognizable in the assembler code is an assignment statement in which an integer constant is moved to a variable. The integer constant appears in the assembler MOV statements, so is easily spotted.

DEBUG is much less useful for debugging background device drivers and interrupt handlers since it is not possible to get control at a breakpoint in the background. DEBUG can still be used to trace through such a device driver outside of PC-McIDAS, though. Suppose you want to trace the interrupt handler for INT xxH. Invoke DEBUG without specifying a file name; i.e.

```
DEBUG
```

Then use the 'A' command to assemble an INT xxH instruction. Use 'T' to execute that instruction. You are now in your interrupt handler. Set the registers to the values expected by the handler, and proceed.

Getting Trace Output Via a Serial Port

It is often the case that one has to debug conditions in real-time. Just tracing through a program in isolation is not good enough. For such cases, a useful technique is to output trace text at 9600 baud to a Televideo monitor. Many of the device drivers already contain code to enable such a trace. They use various conditional assembly flags to determine whether the trace is enabled or not.

The trace code assumes the monitor is attached to serial port 1. To initialize the port, enter the following DOS command:

```
MODE COM1:96,e,7,1
```

Sometimes I have found it helpful to output to a 4800 baud serial printer instead. That way I can pour over the trace output at my leisure.

One has to use some care in tracing from background device drivers. Bugs that involve timing-dependent interactions among background processes may change their behavior if too much trace output is produced. I have found it extremely helpful in such cases to do something like the following.

Suppose some background process is crashing, and you are not sure who the culprit is. Have each suspect process output a single character upon entering, a different character upon leaving. What I usually do, for example, is output a lower case character upon entry ('v' for VIDEO.EXE, for example) and the same character in upper case upon leaving.

Such a trace is usually fast enough not to disturb the condition you are trying to examine, and it lets you determine in which process the crash is occurring. This tells you were to start looking. Note that the trace routines also allow you to output the contents of registers or variables or arbitrary strings of data.

To trace from a Fortran program, use the following entry point:

```
SUBROUTINE SERIAL(CTEXT,IVALUE)
```

CTEXT is a text string terminated by a single dollar-sign. IVALUE is an integer displayed after the text; IVALUE is displayed even if it is 0.

Miscellaneous Tools

Naturally, there are a number of bugs for which DEBUG and serial traces are not the answer. There are two In-Circuit debuggers for AT's at SSEC. I have found them to be indispensable at times.

One often needs to know the scancode associated with a particular key. There is a program SCANCODE.EXE that displays the scan code and ASCII code for any keystroke. Use CTRL-C to get out of SCANCODE.

One often wants to inspect or modify the contents of SYSCOM from within PC-McIDAS. Use the PC-McIDAS commands **LOOK** and **POKE**.

APPENDIX -- SYSCOM DEFINITION

BLOCK 0: Terminal Control Block

Bytes	Item
0-1	terminal number
2-9	8-char terminal id
10	data tablet
11	joystick
12	mouse
13	printer
14	touchscreen
15	monochrome display
16	lo-res graphics display
17	hi-res graphics display
18	SSEC video terminal
19	number of fixed disks in PC
20	computer type (1=AT,2=PS/2)
21	not used
22	flag=1 means RESET command must be run to reset async COMM
23	tv control interrupt rate (ints per second)
24	plus-key toggle type 0 -- toggle inactive 1 -- toggle between text windows and soft tablet 2 -- toggle among text, tablet, and EGA imagery 3 -- toggle between text and EGA imagery

The TCB is set up for 3 (logical) graphics devices and 1 video device. Graphics device 1 refers to graphics on the video device.
Almost all applications programs will deal only with device 1.
(An exception is the user interface subsystem.)

(Each device has a device type code. The following device types are defined currently:
0 -- no device
1 -- [illegible] video device
2 -- [illegible] Graphics Adapter
3 -- [illegible]

APPENDIX -- SYSCOM DEFINITION

BLOCK 0: Terminal Control Block

Bytes	Item
0-1	terminal number
2-9	8-char terminal id
10	data tablet
11	joystick
12	mouse
13	printer
14	touchscreen
15	monochrome display
16	lo-res graphics display
17	hi-res graphics display
18	SSEC video terminal
19	number of fixed disks in PC
20	computer type (1=AT,2=PS/2)
21	not used
22	flag=1 means RESET command must be run to reset async COMM
23	tv control interrupt rate (ints per second)
24	plus-key toggle type 0 -- toggle inactive 1 -- toggle between text windows and soft tablet 2 -- toggle among text, tablet, and EGA imagery 3 -- toggle between text and EGA imagery

(The TCB is set up for 3 (logical) graphics devices and 1 video device. Graphics device 1 refers to graphics on the video device.
Almost all applications programs will deal only with device 1. An exception is the user interface subsystem.)

(Each device has a device type code. The following device types are defined currently:

- 0 == no device
- 1 == SSEC/Dataram video device
- 2 == IBM Color Graphics Adapter
- 3 == IBM Enhanced Graphics Adapter or Video Graphics Array)

Display device 1 (video device):

25	device type
26	number of graphics frames
27-30	total space reserved for graphics frames (bytes)
31-32	start segment of reserved space
33-34	start segment of graphics video RAM for device -- even lines
35-36	start segment of graphics video RAM for device -- odd lines
37-38	bytes per line
39	max graphics color
40-41	lines per graphics frame
42-43	elements per graphics frame
44-47	bytes per graphics frame
48	current graphics frame
49	current graphics mode
50	color palette
51	background color
52	default line width
53	default dash length
54	default gap length
55	default gap color
56	flip flag
57	draw flag
58	graphics screen height (units=0.1 inch)
59	graphics screen width (units=0.1 inch)
60-61	cursor size (vertical)
62-63	cursor size (horizontal)
64-65	cursor position (line)
66-67	cursor position (element)
68	cursor type (1=box, 2=xhair, 3=box&xhair, 4=solid box, 5=star wars)
69	cursor color
70-71	2nd cursor size (vertical)
72-73	2nd cursor size (horizontal)
74-75	2nd cursor position (line)
76-77	2nd cursor position (element)
78	2nd cursor type
79	2nd cursor color
80	cursor mode (0=single cursor, 1=dual cursor)
81	number of image frames
82-85	total space reserved for image frames (bytes)
86-87	start segment of reserved space
88-89	start segment of image video RAM for device -- even lines
90-91	start segment of image video RAM for device -- odd lines
92-93	bytes per line

94	max image color
95-96	lines per image frame
97-98	elements per image frame
99-102	bytes per image frame
103	current image frame
104	dual channel video display flag (0=disabled, 1=enabled)
105	flag=1 means VGA
106-114	not used

Graphics device 2:

115	device type
116	number of graphics frames
117-120	total space reserved for frames (bytes)
121-122	start segment of reserved space
123-124	start segment of graphics video RAM for device -- even lines
125-126	start segment of graphics video RAM for device -- odd lines
127-128	bytes per line
129	max color
130-131	lines per frame
132-133	elements per frame
134-137	bytes per frame
138	current frame
139	current mode
140	color palette
141	background color
142	default line width
143	default dash length
144	default gap length
145	default gap color
146	flip flag
147	draw flag
148	image height (units=0.1 inch)
149	image width (units=0.1 inch)
150-151	cursor size (vertical)
152-153	cursor size (horizontal)
154-155	cursor position (line)
156-157	cursor position (element)
158	cursor type (1=box, 2=xhair, 3=box&xhair, 4=solid box, 5=star wars)
159	cursor color
160-161	2nd cursor size (vertical)
162-163	2nd cursor size (horizontal)
164-165	2nd cursor position (line)
166-167	2nd cursor position (element)
168	2nd cursor type

169	2nd cursor color
170	cursor mode (0=single cursor, 1=dual cursor)
171-184	reserved

Graphics device 3:

185	device type
186	number of graphics frames
187-190	total space reserved for frames (bytes)
191-192	start segment of reserved space
193-194	start segment of graphics video RAM for device -- even lines
195-196	start segment of graphics video RAM for device -- odd lines
197-198	bytes per line
199	max color
200-201	lines per frame
202-203	elements per frame
204-207	bytes per frame
208	current frame
209	current mode
210	color palette
211	background color
212	default line width
213	default dash length
214	default gap length
215	default gap color
216	flip flag
217	draw flag
218	image height (units=0.1 inch)
219	image width (units=0.1 inch)
220-221	cursor size (vertical)
222-223	cursor size (horizontal)
224-225	cursor position (line)
226-227	cursor position (element)
228	cursor type (1=box, 2=xhair, 3=box&xhair, 4=solid box, 5=star wars)
229	cursor color
230-231	2nd cursor size (vertical)
232-233	2nd cursor size (horizontal)
234-235	2nd cursor position (line)
236-237	2nd cursor position (element)
238	2nd cursor type
239	2nd cursor color
240	cursor mode (0=single cursor, 1=dual cursor)
241-252	reserved

253	current relative image frame

254	current relative graphics frame
255	logical device currently owning data tablet's shared physical device
256	default graphics device (which logical device is referred to by commands unless the GDEV= keyword is present)
257	flag=1 means INITSYS.DAT was absent when McIDAS came up so we need to do a full workstation initialization next time a user logs on
258	scheduler counter used to initiate PCCLOC command
259-260	counter for timing scheduler
261-262	count limit after which schedule is checked and counter is reset
263	terminal is remote (=1), local (=0)
264	terminal is video (-1), nonvideo (=0)
265	flag=1 means tvctrl calls mouse interrupt
266	ok for COMM to write to screen (=1) or not ok (=0)
267	tv control tick-counter (used by COMM to control idle messages)
268	flag=1 indicates DOS function is in progress (see DOSFUNC.ASM) (see byte 345 also)
269	flag=1 indicates tv control should stop eating keyboard chars (used to allow applications to read from keyboard directly)
270	j-toggle (connect graphics to loop control)
271	k-toggle (image frame visible/blank)
272	l-toggle (looping on/off)
273	n-toggle (pseudo-color on/off)
274	p-toggle (connect joys to cursor position control)
275	v-toggle (loop velocity cursor)
276	w-toggle (graphics frame visible/blank)
277	y-toggle (connect frames to loop control)
278	z-toggle (connect joy1 to size control)
279	o-toggle (0=image frame loop in force; 1=oppos loop in force)
280	m-toggle (link mouse to cursor)
281-287	reserved
288	single-letter command entered without ALT key
289	who owns the cursor (0=mouse,1=joystick)
290-291	joy1 position (line)
292-293	joy1 position (element)

294-295	joy2 position (line)
296-297	joy2 position (element)
298	joy1 flag (0=disconnected, 1=controls cursor position, 2=vernier size control, 3=controls cursor size, 4=velocity cursor)
299	joy2 flag (0=disconnected, 1=controls cursor position, 2=vernier size control, 3=controls cursor size, 4=velocity cursor)
300-301	unused
302	data tablet shares physical device with other functions
303	data tablet is currently displayed
304	when tablet displayed, logical device which previously owned device
305	flag set to 1 when LOGON sent to host, cleared when PCCLOC runs on AT.. used to let AT know when host logon has completed so AT can proceed with commands in initialization string table, etc.
306-307	mouse line
308-309	mouse element
310	mouse active
311	mouse: left button pushed
312-313	mouse: vertical position when left button pushed
314-315	mouse: horizontal position when left button pushed
316	mouse: right button pushed
317-318	mouse: vertical position when right button pushed
319-320	mouse: horizontal position when right button pushed
321	mouse: both buttons pushed
322-323	mouse: vertical position when both buttons pushed
324-325	mouse: horizontal position when both buttons pushed
326	mouse: cursor visibility
327	mouse: cursor type
328-331	reserved
332	flag=1 means menu system in use
333	flag=1 means we are ready to accept graphics/image packets from host
334	flag used by async COMM to decide when to start accepting packets after init
335	tv control tick-counter (used by COMM to control xon messages)
336-337	touchscreen position (vertical)
338-339	touchscreen position (horizontal)
340	flag=1 means log all commands for UNIDATA workstation

341	flag=1 means comm has timed out for UNIDATA broadcast
342	flag=1 means disable comm timeout checking for UNIDATA broadcast
343	flag=1 means echo command being sent to host (for debugging)
344	flag=1 indicates frame numbers should not be displayed
345	flag=1 means COMM needs to do a DOS function (see byte 268 also)
346-7	tick counter for UNIDATA workstations to signal COMM timeout
348	data tablet pen state (0=up, 1=down)
349	data tablet pen proximity state (0=pen not near tablet, 1=pen near)
350-351	data tablet max x coord + 1
352-353	data tablet max y coord + 1
354	data tablet -- tv space mode
355	data tablet -- inactive area (border) around outside
356	data tablet -- cursor following state
357-358	data tablet -- lower left corner of tv space (line)
359-360	data tablet -- lower right corner of tv space (element)
361	data tablet -- when to start significant event
362	data tablet -- what type of event to start
363	file handle for graphics packets queue
364-5	head of graphics packets queue
366-7	tail of graphics packets queue
368	unused
369	semaphore used to indicate if a command is running...each time a program is spawned, the semaphore is incremented...each time a program finishes, the semaphore is decremented...used to prevent COMM from opening a file while a command is running...otherwise, file will be closed when command completes
370	flag=1 means left mouse button activates user interface from scanner
371	flag=1 means tv control should NOT call text window handler
372	flag=1 means BIOS video function should be intercepted
373	flag=1 means send debugging text to serial port
374	flag=1 means text window handler should NOT display window and comm should not write to screen...flag is

	semaphore used by video int to prevent text window from switching while video int is in midst of writing to window
375	flag=1 means output halted by control-S
376	message is waiting to be transmitted
377-380	address of message buffer
381-382	node address of destination
383	COMM method in use (0=Standalone, 1=Pronet, 2=SNA, 3=Phone,4=Satellite)
384	flag=1 means command or scanner waiting to hear if message was sent ok
385-386	node address of host
387	flag=1 means COMM is temporarily down
388-389	workstation's node address
390	baud rate (1=110,2=150,3=300,4=600,5=1200, 6=2400,7=4800,8=9600,9=19200)
391	parity checking (1=no,2=even,3=odd)
392	data bits (1=7 bits,2=8 bits)
393	stop bits (1=1 bit,2=2 bits)
394	int mask for 8259-1 int controller
395	int mask for 8259-2 int controller
396	flag=1 means COMM should not receive data...some other process wants to intercept it
397	comm port used by async comm
398-399	counter for Ctrl-S timeout
400-401	graphics page boundary: left
402-403	graphics page boundary: right
404-405	graphics page boundary: top
406-407	graphics page boundary: bottom
408	file handle for thread 1
409	file handle for thread 2
410	file handle for thread 3
411	number of open LW files
412-428	palette for EGA hi-res modes
429	flag=1 means text window interface should not echo command line
430-433	segment:offset of BIOS keyboard handler
434-437	segment:offset of KBIOSF keyboard filter routine

438-447	unused

BLOCK 1: Looping Control Block

0-255	primary frame number array (indexed by current relative image frame in TCB)
256-511	graphics frame number array (indexed by current relative graphics frame in TCB)
512-767	opposite frame number array (indexed by current relative image frame in TCB)
768-1023	number of ticks to delay before next step (indexed by current relative image frame in TCB)

The first byte in each array is the number of entries in the array. The succeeding bytes each contain a frame number -- the number of the frame to be displayed when the relative frame pointer points to that place in the array. A -1 entry implies end-of-list.

BLOCK 2: Applications Data Interchange Block

Defined as needed a la positive UC.

0	flag=1 indicates user logged on to PC
1	flag=1 indicates user logged on to host (perhaps unsuccessfully)
2	flag used by GKS to indicate world coords = device coords
3	flag=1 means previous command was PROMPT
4	flag=1 means command line editor is in INSERT mode
5	unused
6-9	address of scanner's copy of COMMON block LBCOM1
10-13	address of scanner's copy of COMMON block LBCOM2
14	flag=1 if COMM is writing a line on screen
15	flag=1 means current command was initiated via a function key

16	flag=1 means RSTI failed to find file to restore...used by IGTV, for example, to decide whether to redraw graphic...
17	set to 1 by "G" key
18	set to 1 by "Q" key
19-22	user's initials
23-24	cursor line for TABWRM
25-26	cursor element for TABWRM
27	EXIT flag
28	restore data tablet label's color when tablet is returned to screen
29-30	label's line
31-32	label's element
33	label's height
34-45	label string
46	length of label string
47	label's color
48	auto-context table flag
49-52	current nav file #
53-56	current MD file #
57-60	current grid file #
61-62	project number
63-70	software release level
71-72	byte number of head of command queue file
73-74	byte number of tail of command queue file
75-76	unused
77	flag=1 means a command is running ... used by UNIDATA workstations to decide when to display "Please stand by..." message
78-79	segment address of tables used by P interrupt
80-81	segment address of tables used by V interrupt
82	flag=1 means COMM has command for scanner to run
83-242	command passed by COMM to scanner
243	flag=1 means user interface has command for scanner to run
244-403	command passed by user interface to scanner
404	semaphore to indicate a dial-in product is being received
405-564	command to be run when a flush is received and semaphore is clear
565-568	local latitude
569-572	local longitude

573-576	local WX station ID

BLOCK 3: Keyin Parameter-Passing Block

0	NFOUND....number of tokens
1	NKEYW....number of keywords
2-65	NARR...number of tokens per keyword (64 x 1 byte)
66-833	CTOK(12,64)....the tokens (64 x 12 bytes)
834-837	IDEVAL....DEV= settings
838-839	DEFOFF....offset in TCB of data for pertinent graphics device
840	GRINIT flag....1 means graphics memory must be initialized 2 means this is first init....3 means GrInit should do nothing but display 'available memory' message...
841	Batch flag....1 means we are starting a batch file
842-853	batch file name
854-861	addresses (far) of two arrays passed by ISQW
862-869	name of SQW'ed program
870-871	PSP segment of SQW'ed program
872-873	data segment of SQW'ed program
874-875	number of local commands
876-3875	names of local commands (CHARACTER*6(500))

BLOCK 4: User Interface Block

1	flag=1 disables toggle out of frame display
2-3	segment for window work area (window to show on next tick)
4	window number displayed on previous tick
5	window number for work seg
6	active window number for BIOS
7	flag=1 means need to re-echo command (e.g. after just switching back to window after displaying frame)
8	for EGA-based system: graphics page currently displayed
9-168	command text for command currently being entered
169	window for text

170	EGA state: 0=text, 1=tablets, 2=frames
171	flag=1 means UNIDATA menu in place...don't write to menu windows
172	color for text
173	color in attribute byte form
174	flag=1 means '+' key has been hit
175	flag=1 means TABLET program is active
176	flag=1 means data tablets are visible
177	flag=1 means menu interface cannot build commands
178	depth of command stack
179	current position in command stack
180-191	string table name for currently active data tablet
192-203	string table name for data tablet window 0
204-215	string table name for data tablet window 1
216-227	string table name for data tablet window 2
228-239	string table name for data tablet window 3
240-251	string table name for data tablet window 4
252-263	string table name for data tablet window 5
264-275	string table name for data tablet window 6
276-287	string table name for data tablet window 7
288-299	string table name for data tablet window 8
300-311	string table name for data tablet window 9
344	flag=1 means scanner does not accept commands from keyboard...used by UNIDATA workstations to restrict input to function keys...also implies text from host not echoed...
345	lowest text window number used by UNIDATA menus
346	current tablet number
347	color for echoing commands
348	color for error messages
349	color for 'Done' messages
350-749	table for setting commands' window, color, and clear flag (50 entries; entry format= command name (6 bytes) window (1 byte) color (bits 0-3) blink (bit 4) mode (bits 5-7)
750-751	mouse line
752-753	mouse element
754	mouse active

755	mouse: left button pushed
756-757	mouse: vertical position when left button pushed
758-759	mouse: horizontal position when left button pushed
760	mouse: right button pushed
761-762	mouse: vertical position when right button pushed
763-764	mouse: horizontal position when right button pushed
765	mouse: both buttons pushed
766-767	mouse: vertical position when both buttons pushed
768-769	mouse: horizontal position when both buttons pushed
770	left mouse button tick counter
771	right mouse button tick counter
772-775	start address of text windows in extended memory
776-779	start address of frames in extended memory (for EGA-based systems)
780	flag=1 means user wants to use ENTER key as line feed
781	flag=1 means ENTER/line feed has been entered
782	flag=1 means keystroke came from Fkey or batch file

BLOCK 5: Voice Interface Block

0	flag=1 means voice interface has a command ready
1-161	buffer for voice interface command
162-383	to be defined

BLOCK 6: Command Stack Block

0-1599	Last 10 commands entered in current session

BLOCK 7: Frame Palette Block

If EGA:

0-15	Palette for frame 1
16-31	Palette for frame 2
.	
.	
.	
240-255	Palette for frame 16

If VGA:

0-47	Color regs for frame 1
48-95	Color regs for frame 2
.	
.	

.

720-767	Color regs for frame 16

BLOCK 8:	COMM File Pool Block
0-4	flag=-1 means corresponding file available for use...flag > 0 indicates command waiting...take highest numbered command first
5-9	file handles for 5 pre-opened temp files used by COMM
10-22	name of 1st file (followed by null)
23-35	name of 2nd file (followed by null)
36-48	name of 3rd file (followed by null)
49-61	name of 4th file (followed by null)
62-74	name of 5th file (followed by null)
75-234	command to unravel 1st file
235-394	command to unravel 2nd file
395-554	command to unravel 3rd file
555-714	command to unravel 4th file
715-874	command to unravel 5th file