

PS 390 DOCUMENT SET

PS 390 ADVANCED PROGRAMMING 1-9

The contents of this document are not to be reproduced or copied in whole or in part without the prior written permission of Evans & Sutherland. Evans & Sutherland assumes no responsibility for errors or inaccuracies in this document. It contains the most complete and accurate information available at the time of publication, and is subject to change without notice.

PS 300, PS 330, PS 340, PS 350, PS 390 and Shadowfax are trademarks of the Evans & Sutherland Computer Corporation.

Copyright © 1987
EVANS & SUTHERLAND COMPUTER CORPORATION
P.O. Box 8700, 580 Arapeen Drive
Salt Lake City, Utah 84108

PS 390 ADVANCED PROGRAMMING

EVANS & SUTHERLAND

The contents of this document are not to be reproduced or copied in whole or in part without the prior written permission of Evans & Sutherland. Evans & Sutherland assumes no responsibility for errors or inaccuracies in this document. It contains the most complete and accurate information available at the time of publication, and is subject to change without notice.

PS 300, PS 330, PS 340, PS 350, and PS 390 are trademarks of the Evans & Sutherland Computer Corporation. DEC, VAX, UNIBUS, and ULTRIX are trademarks of Digital Equipment Corporation. UNIX is a trademark of Bell Laboratories. IBM VM/SP and IBM MVS/TSO are trademarks of International Business Machines.

LIMITED SUPPORT DISCLAIMER

The User-Written Functions facility is distributed by Evans & Sutherland as a convenience to customers and as an aid to understanding the capabilities of the PS 390 graphics systems. Evans & Sutherland Customer Engineering supports the User-Written Functions facility and files necessary to use this facility to the following extent: E&S may supply one or two software packages that are necessary to support user-written functions. The support of these packages is described below.

1. E&S provides two library files USERSTRUC.PAS and USERLINK.ASM and command files to compile, assemble, and link user-written functions with the functions and procedures in the standard PS 390 system. E&S also provides utilities to build a file containing S-Records in a form suitable for downloading to the PS 390 and to load the user-written function via the RS-232 Async interface. These files, any features provided in the PS 390 Graphics Firmware that support user-written functions, and the documentation are all fully supported.
2. E&S may also supply the executable code for the cross-software (compiler, assembler, and linker) that is used to produce the S-records that will be transported to the PS 390. If licensed and purchased through E&S, the executable files will be distributed on a separate tape. If the customer buys a license from E&S, no support will be provided by Motorola. Any problems with the compiler, linker, or assembler may be reported to E&S, but we make no guarantee that they will be fixed. We will answer questions about installing and using these programs, but may refer you to the Motorola documentation for details about the compiler, assembler, and linker. E&S will only deal with questions related to using these programs to write user-written functions. We will not deal with questions relating to how to use the cross-software compiler, assembler, and linker for any other purpose.

CONTENTS

1 THE PS 390 GRAPHICS SYSTEM	1-1
1.1 Hardware Components	1-1
1.2 PS 390 Graphics Firmware	1-2
1.2.1 PS 390 Startup Code	1-3
1.2.2 PS 390 Communications and Graphics Code	1-3
2 MASS MEMORY STRUCTURES	2-1
2.1 Alpha Block	2-2
2.2 Named Entity Block	2-5
2.2.1 Function Instance Block	2-5
2.2.2 Display Structures	2-16
2.2.2.1 Control Blocks	2-16
2.2.2.2 Set Node	2-26
2.2.2.3 Operation Node	2-29
2.2.2.4 Data Node	2-30
2.2.3 Character Font Block	2-34
2.3 Commhead	2-35
2.4 Number Formats	2-35
2.5 Hash Table	2-36

3	INTERNAL PROCESSING	3-1
3.1	Structure Creation	3-1
3.1.1	Alpha Lookup	3-1
3.1.2	Named Entity Creation	3-1
3.1.3	GCP Datum Pointer Setup	3-1
3.1.4	Alpha Update	3-1
3.2	Update Process	3-2
3.2.1	Alpha Update	3-3
3.2.2	Value Update	3-3
3.2.3	ACPProof	3-3
3.2.4	Use of RAWBLOCK	3-4
3.3	Function Operation	3-5
3.3.1	Scheduler	3-6
3.3.2	Function Activation	3-7
3.3.3	Function Status	3-8
3.3.4	Function Code Format	3-9
4	PHYSICAL I/O PROGRAMMING	4-1
4.1	The F:USERUPD Function	4-1
4.2	The Parallel Interface	4-2
4.3	Physical I/O	4-2
4.3.1	Physical I/O Constraints	4-3
4.3.2	Physical I/O Operations	4-3
4.4	Advanced Physical I/O Programming	4-6
5	USER-WRITTEN FUNCTIONS TUTORIAL	5-1
5.1	Introduction to User-Written Functions	5-1
5.1.1	Requirements	5-2
5.1.2	Objectives	5-3
5.1.3	Prerequisites	5-3

5.2	Constructing a Simple Function	5-3
5.2.1	Example	5-4
5.2.2	About Messages and Queues	5-8
5.2.3	About Function States	5-9
5.3	Writing Your Own Function	5-11
5.3.1	Exercise	5-12
5.3.2	Feedback	5-12
5.4	Compiling, Linking, and Naming the Function	5-14
5.4.1	Description of Command Files for DEC VAX/VMS and UNIX ..	5-14
5.4.2	DEC VAX/VMS Command File	5-15
5.4.3	DEC VAX/UNIX Command Files	5-16
5.4.4	Instructions for IBM Systems	5-17
5.4.5	Exercise	5-17
5.4.6	Feedback	5-18
5.5.	Transferring the Function to the PS 390	5-18
5.5.1	Using Routing Bytes to Transfer the S-Record File	5-19
5.5.2	Using Graphics Support Routines to Transfer the S-Record File .	5-20
5.5.3	Exercise	5-21
5.5.4	Feedback	5-21
5.6	Instanting the Function	5-22
5.7	Debugging User-Written Functions	5-23
5.7.1	Exercise	5-24
5.7.2	Feedback	5-24
5.8	Conclusion	5-25
6	MORE ADVANCED IDEAS	6-1
6.1	Example I – Handling Different Message Types on the Same Queue	6-2
6.2	Example II – SET_CNESS and Private Queues	6-4
6.3	Example III – Variable Number of Input Queues	6-7
6.4	Example IV – User-Defined Qdata Type	6-11
6.5	Conclusion	6-18

7	LOADING AND DEBUGGING USER-WRITTEN FUNCTIONS	7-1
7.1	Loading User-Written Functions From Diskette	7-1
7.1.1	Loading the User-Written Function Into Mass Memory	7-2
7.1.2	Loading the User-Written Function and Creating an Instance	7-3
7.1.3	Conclusion	7-5
7.2	PS 390 Debugger	7-6
7.2.1	Using the Debugger	7-6
7.2.2	Debugger Commands	7-9
7.3	Setting Breakpoints in Your Code	7-26
8	USER-WRITTEN FUNCTION REFERENCE	8-1
8.1	Introduction	8-1
8.2	Message Types	8-2
8.2.1	QDtype and Qdata	8-2
8.2.2	Input Message Pointers	8-4
8.2.3	PS 390 Floating-Point Numbers	8-5
8.3	Topical Listing Of Utility Routines	8-6
8.4	Procedures Provided Via USERLINK	8-10
	CkInputs	8-10
	Char_text	8-10
	CkPrivate	8-11
	CleanInputs	8-11
	Csecs	8-11
	DropMessage	8-12
	FCadd	8-12
	FCdivide	8-12
	FCint2double	8-12
	FCinteger	8-12
	FCmultiply	8-13
	FCnearzero	8-13
	FCp2multiply	8-13

FCround	8-14
FCsqroot	8-14
FCsubtract	8-14
Fpabs	8-14
Fpecomp	8-14
Frames	8-15
HRTIME	8-15
Int_text	8-15
MsgCopy	8-16
My_in_out	8-16
My_name	8-16
Newqboolean	8-16
Newqinteger	8-16
Newqmatrix	8-17
Newqnil	8-17
Newqpacket	8-17
Newqreal	8-17
Newqvector	8-18
Newtry	8-18
QIIMessage	8-18
QIIValue	8-18
Qincompatmsgs	8-19
QSendCopyMess	8-19
Real_text	8-19
Rndmnumber	8-20
SavePrivate	8-20
SendMsg	8-20
Set_Cness	8-21
Sincos	8-21
Systemerror	8-22
Text_text	8-22
Ticks	8-22
Time_text	8-22

	UWError	8-22
	Vfetch	8-23
	Vstore	8-23
8.5	Advanced UWF Procedures	8-24
	Lk_cursuffix	8-24
	Lk_nosuffix	8-25
	Lgaupdate	8-25
	Announceupdate	8-26
	Msgstore	8-26
	Setlock	8-26
	Clrlock	8-27
	Incausage	8-27
	Decausage	8-27
	AcpProof	8-27
	Acpprf1	8-28
	OLbaddtoaset	8-28
	Removefromaset	8-29
	FetchBlock	8-29
	Acp_v3f	8-29
	Acp_v2f	8-30
	Acp_v3b	8-30
	Acp_v2b	8-30
	nStoreVector	8-31
	nNewAcpdata	8-31
	Store3x3	8-31
	Store4x4	8-32
	Drop_name	8-32
	GetVector	8-32
	Rawbacopy	8-33
	Rawcbcopy	8-33
	Rawchcopy	8-33
	Size_of	8-33
	FetchAdnum	8-34

nFetchCopy	8-34
WaitFrame	8-34
loc_thead	8-35
ptr_dcb	8-35
DropNE	8-35
Newreturns	8-35
Reactivate	8-36
Myanyoutputs	8-36
Pushmyinput	8-36
WaitCsec	8-36
HA_cursor	8-37
HA_no_cursor	8-37
8.6 Stack Size	8-38
8.7 Error Messages	8-38
 9 APPENDICES	 9-1
9.1 Using the Command Files on DEC VAX/VMS	9-1
9.2 Using the Command Files on DEC VAX/UNIX	9-11
9.3 Using the Cross-Software on IBM VM/SP	9-18
9.4 Using the Command Files on IBM MVS/TSO	9-24
9.5 USERSTRUCT.PAS	9-25
9.6 Function Header Line Format	9-33
9.7 S-Record Format	9-34
9.8 Motorola Pascal Register Usage	9-37
9.9 Commhead Format	9-41
9.10 Operation and Data Node Formats	9-44
9.11 Error Types/Error Numbers	9-62
9.12 F:USERUPD	9-69

FIGURES

Figure 2-1	Alpha Block and Pascal Data Definition	2-4
Figure 2-2.	Function Instance Data Structures	2-8
Figure 2-3.	Function Instance Block and Pascal Data Definition	2-9
Figure 2-4.	Function Inputs Block and Pascal Data Definition	2-12
Figure 2-5.	Outset Block and Pascal Data Definition	2-14
Figure 2-6a.	DCR Block	2-17
Figure 2-6b.	Pascal Data Definition of DCR Block	2-18
Figure 2-7a.	DCB Block	2-21
Figure 2-7b.	Pascal Data Definition of DCB Block	2-23
Figure 2-8a.	Set Nodes	2-27
Figure 2-8b.	Pascal Data Definition of Set Node	2-28
Figure 2-9.	Operation Node	2-29
Figure 2-10a.	Data Node	2-30
Figure 2-10b.	Pascal Data Definition of Data Node	2-31
Figure 2-11a.	Character Font Blocks	2-34
Figure 2-11b.	Pascal Data Definition of Character Font Blocks	2-35
Figure 3-1.	Rawblock	3-4
Figure 4-1.	Format of Physical Read Address List	4-4
Figure 4-2.	Format of Data From PS 300 in Physical Read	4-4
Figure 4-3.	Format of Data to PS 300 in Physical Write	4-5
Figure 9.10-1.	General Operation Node Format	9-45
Figure 9.10-2.	Viewport	9-45
Figure 9.10-3.	Character Rotate, Character Scale, Character Size, Matrix_2x2	9-45
Figure 9.10-4.	Rotate, Scale, Matrix_3x3	9-46

Figure 9.10-5.	Window, Eye Back, Field_of_View, Matrix_4x4	9-46
Figure 9.10-6.	Translate	9-46
Figure 9.10-7.	Increment Level-of-Detail	9-47
Figure 9.10-8.	Decrement Level-of-Detail	9-47
Figure 9.10-9.	Set Level-of-Detail, Set Conditional Bit, Set Displays, Set Character, Orientation, Set Contrast, Set Depth_Clipping, Set Rate External, Set Blinking, Set Line_Texture	9-47
Figure 9.10-10.	IF Conditional_Bit, IF Phase (Bit 15)	9-49
Figure 9.10-11.	IF Level_of_Detail	9-49
Figure 9.10-12.	Look At/From, Matrix_4x3	9-50
Figure 9.10-13.	Set Picking Location	9-50
Figure 9.10-14.	Set Picking Identifier	9-50
Figure 9.10-15.	Character Font	9-51
Figure 9.10-16.	Set Color	9-51
Figure 9.10-17.	Set Rate	9-51
Figure 9.10-18.	Set Intensity	9-52
Figure 9.19-19.	Xform Matrix	9-52
Figure 9.10-20.	Xform Vector	9-52
Figure 9.10-21.	Writeback	9-52
Figure 9.10-22.	Solid_Rendering, Surface_Rendering	9-53
Figure 9.10-23.	Sectioning Plane	9-53
Figure 9.10-24.	Light Pen	9-54
Figure 9.10-25.	Text Size	9-55
Figure 9.10-26.	Load Viewport	9-55
Figure 9.10-27.	Set Blink Rate	9-55
Figure 9.10-28.	Load Picking Location	9-55
Figure 9.10-29.	General Data Node Format	9-56
Figure 9.10-30.	Vector_List N=n X1,Y1,Z1 X2,Y2,Z2 ... Xn,Yn,Zn Vector-Normalized (Full Vector) - 3D (Vec3f0)	9-57
Figure 9.10-31.	Vector_List N=n X1,Y1,-- X2,Y2,-- ... Xn,Yn,-- Vector-Normalized (Full Vector) - 2D (Vec2f0)	9-57
Figure 9.10-32.	Characters, Labels Character string (DstringD)	9-58
Figure 9.10-33.	Vector_List Block N=n X1,Y1,Z1 X2,Y2,Z2 ... Xn,Yn,Zn	9-59

Figure 9.10-34. Vector_List Block N=n X1,Y1 X2,Y2 ... Xn,Yn	9-59
Figure 9.10-35. Vector_List N=n X1,Y1,Z1 X2,Y2,Z2 ... Xn,Yn,Zn	9-60
Figure 9.10-36. Vector_List N=n X1,Y1 X2,Y2 ... Xn,Yn	9-61

DIAGRAMS AND TABLES

Function States Diagram	5-10
Function Network Diagram 1	7-5
Function Network Diagram 2	7-6
Table 7.2-1 Commands Accessing “Open” Memory Locations	7-9
Table 7.2-2 Commands to List Data in Memory	7-13
Table 7.2-3 Program Execution and Debugging Commands	7-14
Table 7.2-4 Breakpoint-Related Commands	7-16
Table 7.2-5 Hunt Commands	7-18
Table 7.2-6 Boot-related Commands	7-20
Table 7.2-7 Memory Test Commands	7-23

Section AP1

The PS 390 Graphics System

This document is intended to provide information about the data formats and system function of the PS 390. It contains the information necessary to effectively write advanced user-written functions and to perform physical I/O functions across the high-speed parallel interface. The PS 390 is a data-driven, interactive display system. It consists of several general- and special-purpose processors and subsystems that are interfaced by means of a general data bus in conjunction with a mass memory system.

1.1 Hardware Components

The Joint (graphics) Control Processor (JCP) is a general-purpose microprocessor that manages the data structures in mass memory and initiates the display defined by these data structures. The JCP contains 512K bytes of local program memory. The Mass Memory is a general-purpose, dual-ported memory that is byte-addressable by the JCP. Up to 2048K of mass memory is available on the JCP card. The PS 390 can also be configured with up to two additional Mass Memory cards, each of which contains 1024K bytes of memory. Mass memory may be used as program memory for the JCP, but generally provides the memory in which data structures are stored and manipulated by the JCP and accessed for display by the Display Processor. Each refresh cycle, the Display Processor traverses the data structures and transforms the data to be displayed; performs clipping, perspective projections and viewport mapping; and finally draws the data on the CRT. The Display Processor is comprised of an Arithmetic Control Processor(ACP), a Pipeline Subsystem(PLS), a Raster Backend Bit-slice processor(RBE/BS), and a Raster Backend Video Controller(RBE/VC).

- The Arithmetic Control Processor is a special-purpose, bit-slice microprocessor that interfaces with the Mass Memory by means of a high-bandwidth, 32-bit memory access port. The ACP traverses linked data structures in Mass Memory, referred to as SOD (Set-Operate-Data) structures. The SOD structures contain commands that indicate the functions the ACP is to perform for each SOD data block. These include commands to modify the state of the ACP and process three-dimensional data.

The state of the ACP is considered to be those values which are context dependent, such as transformation matrix contents, viewport boundaries, color, line texture, etc. The SOD structures and the commands which initiate and control the ACP functions are detailed in later sections. In traversing the SOD structures, the ACP performs all of the matrix operations required to transform data points before passing the transformed data coordinates to the Pipeline Subsystem.

- The Pipeline Subsystem accepts transformed data coordinates from the ACP and performs clipping, perspective division, and viewport mapping on the data to be displayed. The Pipeline Subsystem processes data asynchronously in relation to the processing performed by the ACP. The processed data is then output to the Raster Backend.
- The Raster Backend accepts screen coordinate data and commands from the Pipeline Subsystem. From this data, it produces pixel information for the display of images on the raster monitor.

1.2 PS 390 Graphics Firmware

The primary purpose of the PS 390 graphics firmware is the manipulation of data structures that are traversed by the Display Processor to display information. To do this, the graphics firmware must:

1. Perform startup operations
2. Communicate with the host computer support software
3. Receive input from graphics peripherals
4. Initiate updates of the data structures
5. Perform data storage allocation/deallocation
6. Initiate the traversal of the data structures by the Display Processor.

The graphics firmware is made up of startup code and communications and graphics code.

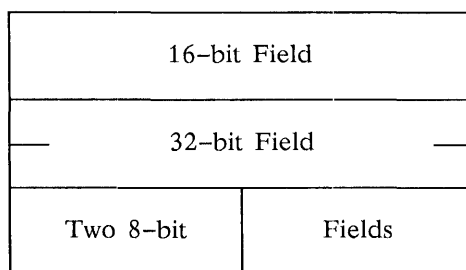
1.2.1 PS 390 Startup Code

This determines hardware availability and status by performing a series of self-tests on hardware components. Following these self-tests, system configuration parameters and the 68000 Graphics Control Program(GCP) are loaded from the PS 390 floppy disk drive. (Throughout this document, GCP is used interchangeably as an acronym for the Graphics Control Program and for the Graphics Control Processor on the JCP card.)

1.2.2 PS 390 Communications and Graphics Code

Once startup code is completed, the Graphics Control Program then initializes all data structures and communications handlers and awaits input from host software or keyboard.

Included in this document are several block diagrams of data structures. The values of fields are indicated graphically, as in the following example:



Section AP2

Mass Memory Structures

PS 390 data structures are built by the GCP and exist in mass memory. Data structures in the PS 390 which can be named by the user are referred to as Named Entities. Named Entity blocks represent function instances, variables, character fonts and display data structures. These types of structures are called Named Entities, even if there is no associated name. Hence, the term Named Entities refers to blocks which can be (but not necessarily are) named. The name associated with a Named Entity is kept in a block referred to as an alpha block. An alpha block is a data structure that contains the location in mass memory of a Named Entity as well as the name, if any, associated with that Named Entity. Every Named Entity, regardless of whether the user has chosen to name it or not, has an alpha block associated with it. Alpha blocks with names are listed in a dictionary that is indexed by means of a fixed-length hash table. The hash table is an array of pointers to forward and backward linked lists of alpha blocks. Naming a node causes the name to be entered into a hash table along with a pointer to where the current node associated with that name resides in PS 390 Mass Memory.

Though the address of the Named Entity referred to may change often, the address of the alpha block remains constant until an INITIALIZE command is entered, which will destroy the alpha block.

In general, all references to a Named Entity structure are indirect, through its alpha block. Whenever some node in the system references another node, a pointer is placed to the alpha block (known as an “alpha pointer”) so that the current node associated with that name can be determined.

2.1 Alpha Block

The alpha block structure is shown in Figure 2-1. Some contents of the alpha block are:

- Datum Pointer

The datum pointer points to the location in Mass Memory of the Named Entity to which the alpha block refers.

- Dictionary Forward and Backward Pointer

The dictionary forward and backward alpha pointers allow an alpha block to be linked into the dictionary list.

- Usage Count

Usage count indicates the current number of references made to the alpha block in the display data structures or function networks. It is necessary since names may be multiply referenced. The usage count must be incremented for every new pointer to this alpha block, and must be decremented each time a pointer to the block is removed. When the usage count becomes zero, the alpha and its associated Named Entity block are removed from storage. Two utility routines, Incausage and Decausage exist to perform these actions.

Note

It Is Very Important To Keep This Count Accurate! If the count is too small, it will crash the system at some later time; if it is too large, the memory will never be recovered.

- Alock

This is a Boolean value that is set true when the alpha is being manipulated, false otherwise. Normally, the utility procedures provided handle locking and unlocking of this field when it is necessary.

- Ci_num

This field is a positive integer which indicates which instance of the Command Interpreter created the alpha block. Each Command Interpreter has associated with it a Ci_num which is the parameter that is given when instantiating the command interpreter (for example, CInstance := f:ci(4);). This number is kept in the CI's private data and is written into the alpha block for each alpha it creates. Note that once an alpha has been created, it “belongs” to this CI until an INIT command occurs. Even redefinition of the contents of an alpha block by another instance of the CI does not change ownership of the alpha. When the CI function instance receives an INITIALIZE command, it can then identify those alphas in the dictionary which it created and which need to be destroyed.

- Gcpdatum

The Gcpdatum points to the location in Mass Memory of the Named Entity to which the alpha will refer after all current data structure updates have been done. Whenever the GCP needs to find the Named Entity which is referred to by an alpha, it must look at this field rather than the Datum field, since the Gcpdatum field has the latest changes. This field disagrees with the Datum field only when an alpha update is pending. See the section on Alpha Updates for further information.

- Namelength

The namelength specifies the byte length of the name associated with the alpha block.

- Name

The name is a string of characters representing the name of the Named Entity.

Datum Pointer	
Dict Forward Alpha Pointer	
Dict Back Alpha Pointer	
Usage	Alock - Ci_num
Gcpdatum	
Reserved	
n = Namelength	Char 0
Char 1	Char 2
• • •	
Char n-1	Char n

TYPE

```
Ptralphablk      = ^ Alphablk;
PtrNamedEntity   = ^ NamedEntity;
Namespell        = ARRAY[1..255] OF Char;
Lock              = Boolean;
Int16             = -32768..32767;
```

```
Alphablk         =
```

RECORD

```
    Datum: Ptrnamedentity;      { Alpha's Named Entity }
    Dictfwd: Ptralphablk;       { Dictionary list pointers }
    Dictback: Ptralphablk;
    Usage: Int16;                { Reference count }
    aLock : Lock ;              { Lock on this alpha }
    Ci_num: Int8 ;               { ID number of creating CI }
    Gcpdatum: Ptrnamedentity;    { Alpha's Named Entity }
                                { after all updates }
    UserDatum: Ptrnamedentity;   { Reserved for future use }
    Namelength: Int16;           { Number characters of name }
    Name: Namespell;             { Alpha's name }
END; { of Alphablk }
```

Figure 2-1. Alpha Block and Pascal Data Definition

2.2 Named Entity Block

A Named Entity is a basic data structure type of the PS 390 system that can be named and referenced. Some of the Named Entity types are as follows:

- **Function Instance**

A function instance is a Named Entity that performs a given function, based upon a given set of inputs, and creates a given set of outputs.

- **Display Structures**

The ACP traverses these each refresh cycle for display purposes.

- **Character Font Block**

A character font block is a structure that defines the strokes or vectors which make up the characters. It consists of an integer that indicates this display structure is a character font block, a 128-entry (or 256) character font table (one for each character of the 128 displayable ASCII character set), and up to 128 (or 256) associated character stroke blocks (with strokes in relative mode).

2.2.1 Function Instance Block

Functions perform specific operations by accepting input, processing it, and sending output to other functions. Generic functions exist as Pascal-callable procedures which reside in the Graphics Control Program. Since each function is guaranteed to run to completion once execution has begun, the module of code for a generic function can be shared among several instances of the function without regard to reentrancy requirements (as long as sufficient mass memory is available). Care must be taken, however, to ensure that all residual data are kept on a private data queue which is accessible only to that function instance. Functions may be used individually or be part of a network of functions to perform a required operation. Three types of functions exist: standard functions, system functions, and I/O functions.

1. **Standard functions**

Standard functions include all user-created functions and those functions receiving input from devices. Standard functions communicate through their inputs and outputs. They have specified priorities and are scheduled for execution accordingly by the Scheduler.

2. System functions

The system functions are special-purpose functions which can be created only by the firmware itself and which cannot be part of a user-created function network. One system function is the DESTROY function, which returns memory blocks to free storage. Its input queue is accessible to all functions, not by a function connection, but by a global variable. Another system function is the UPDATE-FOR-MATTER function, which takes update blocks from a system-wide update list and prepares their contents for the ACP to work on. This function has no inputs or outputs but is activated when an update is put on the update list. The UPDATE-KILLER function also has no inputs or outputs as other functions do. It is activated when update blocks have been processed by the ACP and need to be destroyed.

3. I/O functions

I/O functions perform input/output operations for I/O devices only. One input function and one output function exist per device. I/O functions are unique in that they communicate with interrupt-level routines through special buffers rather than through the normal function communication method of queued messages. Specifically, an input function differs from other functions in that it has no input queues. Instead, its data come from the specific hard-coded buffer associated with its input device. When an input interrupt occurs, an interrupt routine places all input in this buffer and determines if an input function is waiting on that buffer. If so, the input function is activated. An output function is special in that it is the only function which waits on both an input queue and an output “queue”. The output function receives input as queued messages and outputs to an associated hard-coded buffer through a special interrupt procedure. It may have to wait either for room in the output buffer or for new queued messages. Note that because I/O functions can communicate with other functions, they have names which the host can access (for creating function networks), as well as standard communication queues for that purpose.

- Function Instance Data Structures

When the need for a particular function arises, an “instance” of the generic function is created by the user or by the system. This function instance has its own unique set of input sources, output destinations, and private data.

Before a function instance can be created, enough mass memory must be reserved to accommodate the necessary data structures. These data structures are: a Function Instance block, a Function Inputs block, and Outset blocks.

To understand these data structures, it is important to note that PS 390 Pascal is not standard Pascal. In standard Pascal, the length of all records is known at compile time. In PS 390 Pascal, the size of one array in the record can be determined at runtime. Because this array is placed in the last field of the record, the compiler can correctly compute the starting offset of that last field (as well as all prior fields). Standard Pascal allows a similar subterfuge in permitting variant records. With the PS 390 Pascal, note that the variance is extended to include the size of a final array. For example, a variable number of inputs and outputs is allowed for a function instance. The last field in the function instance block is an array (of variable length) of outset pointers (one for each output). In order to allow a variable number of inputs, a separate Function Inputs block was defined whose last field is an array of input queue pointers (one for each input).

The Function Instance structure contains all the references to data needed to define a function instance. This information includes (a) an index identifying the Pascal-callable procedure that performs the required generic function, (b) the input source(s), (c) the output destination(s), (d) a priority value for scheduling purposes, (e) the current status of the function, and other information about the function.

The Function Instance block contains the address of a Function Input block. Each input queue is defined by a Message Queue block. The Function Inputs block contains the number of inputs for the function, a message queue for the private message of the function, and one message queue for each other input of the function.

The Message Queue block references the input data itself. It also indicates if the function instance is waiting for input on its associated data in order to be activated, though this technique is used only if the function requests to wait for a specific input rather than a full set of inputs.

The Function Instance block also points to Function Outset blocks. The Function Outset block identifies each of the output destinations of this function instance, including the appropriate queue of the receiving function or display data structure.

The Function Outset blocks contain two int8's (aside: an int8 implies that the maximum degree of fanout from a function is 127). One int8, Maxn, gives the capacity of the block; the other, N, gives the capacity actually used, where capacity means the number of output designators that will fit in the block.

A negative N is initially indicated. If a function attempts to send data out of an output port with a negative N, a warning "No connection ever made out of ..." (or somesuch) is issued, and the used field is set to zero (this prevents repeated warnings of no connection ever made). An effect of a DISCONN ... :ALL on that output is to set N field to zero; this provides a means of the user specifically to indicate that no connection was intended to be made out of that input and hence disable the warning. The N also only goes down to zero (i.e., is not made negative) if each connection ever made is specifically removed (as opposed to the :ALL technique).

Figure 2-2 illustrates the general relationship between these function instance data structures. Then next sections detail each of these structure blocks.

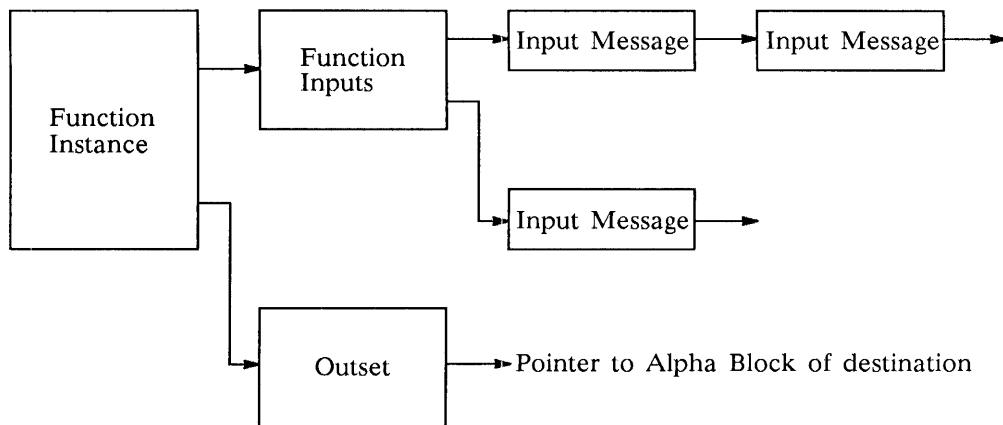


Figure 2-2. Function Instance Data Structures

- **Function Instance Block**

A function instance is represented in storage by a function instance block of the format shown in Figure 2-3.

Function Instance		= 4
Named Entity Pointer		
Function Status	Priority Value	
My Alpha Pointer		
Debug Alpha Pointer		
Debug Alpha Pointer		
Myint8	Carve Memory	
Debug Function Code		
Pascal Function Code		
Function Inputs Pointer		
Num of Outputs	IamMemOK	
NumNonNull		
Outset 1 Pointer		
		•
		•
		•
Outset n Pointer		

```

CONST
    Fcninstance      =      4;
TYPE
    PtrNamedEntity   =      ^ NamedEntity;
    Ptralphablk      =      ^ Alphablk;
    Ptrfcinputs       =      ^ Fcinputs;
    Ptroutset         =      ^ Outset
    Int8              =      -128..127
    Fcninstance       =
    RECORD
        Type: Fcninstance;
        Schednext: PtrNamedEntity;
        Status: Fcn_status;
        Priority: Int8;
        Myalpha: Ptralphablk;
        A_dbgcmdex: Ptralphablk;
        A_dbgprint: Ptralphablk;
        Myint8: int8; {1-byte private message}
        Carvememory: Boolean;
        D_fcncode: Fcntype;
        P_fcncode: Fcntype;
        Inputs: Ptrfcinputs;
        Noutputs: Int8;
        IamMemOK : MemOKindex;
        NumNonNull : Int16;
        Outputs: ARRAY [Dummysize]OF Ptroutset;
    END;

```

Figure 2-3. Function Instance Block and Pascal Data Definition

The fields of the function instance block are detailed below.

- Function Instance

This field identifies this block as a function instance block.

- Named Entity Pointer

The second field in the function instance block is for use by the Scheduler. The Named Entity pointer identifies the function instance next to run on the active or priority function list.

- Function Status

This field indicates the current state of the function instance.

- Priority Value

The priority value gives the priority of the function. The Scheduler uses this value at scheduling time to determine when this function will execute in relation to other functions ready to run. Lowest priority is 15 and the highest is 0.

- Alpha Pointer

The alpha pointer (Myalpha) points to the block in memory containing the user- or system-given name of the function instance. Note that because a function instance is a Named Entity, it may be named by the user. Through this name, the user may create the function instance, delete it, or display data structures.

- Debug Alpha Pointers

These alpha pointers are currently not used.

- Myint8

This field is used by individual functions for private data. For example, some of the data concentrator functions indicate which port this function serves via this field.

- Carve Memory

This is a boolean which indicates whether this function might need to carve memory.

- Debug Function Code

This field identifies the generic function (a Pascal-callable procedure) to be called when the function instance is executed.

- Pascal Function Code

This code identifies the generic function (a Pascal-callable procedure) to be called when the function instance is scheduled for execution. (This field is always the same as the Debug Function code.)

- Function Inputs Pointer

The function inputs pointer points to a function input block that lists the function instance inputs.

- Number of Outputs

Number of outputs specifies how many different output values are to be sent from this function to other functions.

- IamMemOK

This is index into array of users. Free storage used by this function will be charged to the user according to this index. Also, the scheduler uses this index to determine which user category this function is in.

- NumNonNull

This represents the number of inputs queues which still need data in order for this function to be able to run. When a message is sent to a previously empty input queue, this number is decremented; when it becomes 0, the function is ready to be executed.

- Outset Pointers

Each function output port has an outset block. The outset block identifies output destination(s).

- Function Inputs Block

A function input arrives in the form of a queued message. A function inputs block, see figure, contains the input queue headers.

Number Inputs	Not Used
Private Queue Head Pointer	
Private Queue Tail Pointer	
Cqueue	Ownerwait
Input 1 Queue Head Pointer	
Input 1 Queue Tail Pointer	
Input 1 Cqueue	Ownerwait
	•
	•
	•
Input n Queue Head Pointer	
Input n Queue Tail Pointer	
Input n Cqueue	Ownerwait
Reserved	

TYPE

```

Mqueue      =
RECORD
    Qhead: Ptrqdata;           {head of queue}
    Qtail: Ptrqdata;           {tail of queue}
    CQueue : boolean;          {true - Cqueue, false - Tqueue}
    Ownerwait : boolean;       {true function waiting on this}
                                {queue for activation}
END; {of Mqueue}
Fcinputs    =
RECORD
    Nin: Int8;                 {number of input queues}
    Private: Mqueue;           {private data}
    Inputqueues: ARRAY[Dummysize] OF Mqueue; {actual input queues}
END; {of Fcinputs}

```

Figure 2-4. Function Inputs Block and Pascal Data Definition

The fields of the function input block are detailed below.

- Number of Inputs

Number of inputs specifies the number of (public, not private) input queues of the function instance.

- Private Message Queue Head and Tail Pointers

A private data queue retains variables needed from one execution of the function instance to the next. Private message queue head and tail pointers point to the first and last messages on this queue.

- Inputqueues Head and Tail Pointers

These pointers point to the first and last messages on this queue.

- Cness

This boolean value indicates if this queue is to be treated as a Constant or Trigger queue. Each function is initialized with specific defaults for each queue. This value may be changed from its default via the SETUP CNESS command.

- Ownerwait

This indicates if the function is waiting for input on this queue. It is only used if the function requests to wait on this specific queue. Most functions do not make use of this field anymore.

- Outset Block

An outset block, shown in Figure 2-5, gives the set of destinations for an output of a function. Each different output has an outset block. An outset block may specify any number of function instance or display data structure recipients; each is identified by its alpha pointer. The input number designates which input queue of the destination function or structure will receive this function's output.

- Qdata Blocks

A function's input queues are queues of Qdata blocks, each containing a message from another function. A message can communicate either data or an event (or both), and is one of many types. Each Qdata block has a pointer to the next message on the queue, a field indicating the message type, followed by the message itself, whose contents and format depend upon the message type. Note that some Qdata are special types put on a private queue. A private data queue retains variables needed from one execution of a function instance to the next.

- Function Instancing

The function instance data structures provide the means for interfunction communication. Most commonly, function instances receive inputs from and send outputs to other function instances within a “function network.” Function instancing is the process of creating a function instance and tying it to the existing function network. Some function instances are instanced automatically in the PS 390 initialization code; others are done at the user’s command.

Number Recipients	Number this block
Alpha Pointer 1	
Input Number 1	
Ci_num	Memok
Alpha Pointer 2	
Input Number 2	
Ci_num	Memok
• • •	
Alpha Pointer n	
Input Number n	
Ci_num	Memok

TYPE

```
Ptralphablk = ^Alphablk;
Int8         = -128..127;
Int16        = -32768..32767
```

```
Outdesignator =
```

```
RECORD
```

```
  Who: Ptralphablk;           {Alpha of destination}
  n: int16;                   {index into that block}
  CI_num : int8;              {CI that did the connect}
  memok : memokindex         {user classification of the connection}
```

```
END; {of Outdesignator}
```

```
Outset =
```

```
RECORD
```

```
  n: Int8;                    {set of output designators}
  Maxn: Int8;                 {number of recipients}
  o: ARRAY[Dummysize] OF Outdesignator; {# of Outdesignators this block can hold}
```

```
END; {of Outset}
```

Figure 2-5. Outset Block and Pascal Data Definition

To create a function instance, the system does the following:

1. Obtains a function instance block for the function and loads it with the function code and priority values.
2. Assigns to the instance the user- or system-specified name (in an alpha block), ties the alpha into the name dictionary (if it is not already there), and enters the alpha address into the function instance block.
3. Sets up the input, output, and private data queue structure. This involves creating a function inputs block, and an outset block for each output, then tying the blocks together in the proper manner.
4. Executes initialization code for the function, if any exists.
5. Marks the function's state as "Act on update" and then asks the ACP to associate the function with its name. (Some functions created at boot time are just directly associated. This is safe since the ACP cannot be using that name yet since the ACP is not running).
6. When the update killer function in the GCP sees that the name association for a function has been performed, it activates the function. The next time the scheduler runs, it will place the function on its appropriate active list.
7. Executes the function. During this first execution, the function causes itself to wait for data in the appropriate manner (i.e., from a queue, from the clock, or from a device). In addition, it may perform some initialization specific to that function (such as setting up its private queue data), although this is usually done in the function initialization code described above.

Once a function instance has been created, the user or system can connect it into the existing function network by issuing a command to tie the function's outputs to their destinations. This is done, for any given output, by entering into the outset block a pointer to and an index into the receiving function instance or data structure node. In the same manner, other functions may be connected to this function through its alpha (or name). Note that since connections to receiving functions (and display structures) are made via their alphas, those functions (and structures) need not exist prior to this connection, and a redefinition of the receiving function does not destroy connections from other functions into it.

2.2.2 Display Structures

Display structures in the PS 390 represent the operations and data that form the two- or three-dimensional objects constructed by the user. The objects that are formed are represented by a structured display file that is traversed each refresh cycle by the ACP. The structured display file is created and modified under control of the Graphics Control Processor (the 68000 in the system). The elements of the structured display file are quite simplistic in nature, forming an acyclic graph of nodes that are either:

- Control blocks (DCR, DCB)
- Data (dots, lines, or characters)
- Operations that change the “state of the machine” for descendent data nodes.
- Set nodes that contain lists of branches of the acyclic graph that are to be traversed.

With the exception of the control blocks, these nodes are all Named Entities, thus they each have an associated alpha block which may include a name.

2.2.2.1 Control Blocks

The Display Control Root (DCR) and Display Control Block (DCB) are control blocks which serve as the major communication links between the ACP and the GCP. The DCR contains the items used for communication between the GCP and ACP. The DCR contains the address in Mass Memory where the first DCB resides. A DCB is the topmost node of the user’s structure; one DCB exists for each user.

- Display Control Root (DCR)

The Display Control Root (DCR) is a block of storage that serves as the root node of the display data structures. The DCR, in conjunction with the Display Control Blocks, is the only means by which the GCP and the Display Processor (ACP) communicate directly.

During system initialization, the GCP writes the address of the DCR to the ACP. The ACP microcode can then read this address and store it

internally. The ACP controls the refresh synchronization by means of the DCR. The ACP polls a flag in the DCR, waiting for the flag to be set by the GCP to initiate a refresh cycle. When the GCP sets the flag, the ACP begins traversing the display data structures. When the ACP completes traversing the data structures, it performs the updates specified by the DCR, resets the flag, and restarts the process by polling the DCR again and then waiting for the GCP to initiate another refresh cycle. The DCR is shown in Figure 2-6.

Display Processor Busy Flag
Do Updates Flag
Plot Done Flag
Plot Select
Upblock List Head
1st DCB Pointer
Transmit Trigger
Progress Flag
X Start
Y Start
////////////////////////////////
Transmit Mode
FIFO Head
FIFO Tail
////////////////////////////////
ACP Dispose Flag
////////////////////////////////
Free Storage Size
Free Storage Pointer
Timeout Delta
ACP Status
Upblock List Tail
Timeout Function Pointer
ACP Free Owner
ACP Free Lock //////////////////////////////////
Last Shade Block

Figure 2-6a. DCR Block

```

TYPE
  Int16          = -32768..32767
  Bitmask        = Int16;
  PtrAvupblock   = ^Avupblock;
  PtrDCB         = ^DCB;
  Ptrnamedentity = ^Namedentity;
  Lock           = BOOLEAN;

  DCRblk         =
    RECORD
      ACPbusy: Int16;
      Douupdates: Int16;
      Plotdone: Bitmask;
      Plotsselect: Bitmask;
      Avup: PtrAvupblk;
      FirstDCB: PtrDCB;
      Xmit_Trigger : Int16;
      Progress_flag : Int16;
      X_Start : Int16;
      Y_Start : Int16;
      unused0 : Int16;
      Xmit_mode : Int16;
      FIFO_head : Ptrqdata;
      FIFO_tail : Ptrqdata;
      unused1 : Int16;
      ACPDisposeFlag : Int16;
      unused2 : Int16;
      ACPFreeSize : Integer;
      PtrACPFree : PtrIntArray;
      TimeoutDelta : Int16;
      ACP_Status : Int16;
      Avupt: Ptravupblk;
      Timeoutfcn: Ptrnamedentity;
      ACPFreeowner : Ptrnamedentity;
      ACPFreeLock : Lock;
      LastshadeBlock : Ptrqdata;
    END;

```

Figure 2-6b. Pascal Data Definition of DCR Block

As the figure shows, the DCR consists of several items, ten of which are explained below:

- ACP Busy Flag

The Display Processor busy flag (ACPbusy) is the location in Mass Memory that the ACP polls to determine when to begin traversing the

display data structures. The GCP sets this location to a non-zero value to initiate a new refresh cycle, synchronizing the setting of this location with the line frequency. When the ACP has completed its traversal (and update) cycle, it zeroes this location, indicating it is idle again.

- Do Updates Flag

The Do Updates flag (Doupdates) is a location that the ACP examines at the end of each refresh cycle to determine whether updates are to be performed. If this location is non-zero, then the update list head must be examined to determine the address of the first update to perform.

- Plot Done Flag

This field is not used in the PS 390.

- Plot Select Flag

This field is not used in the PS 390.

- Update List Head and Update List Tail

When a change needs to be made in the display data structures, the GCP communicates the change to the Display Processor in an update block (Avupblk). The update block is on an update list, which has a head and tail pointed at by the DCR update list head and list tail. The update list head indicates the address of the first update the ACP is to perform. Once an update cycle is completed, the ACP zeroes the (Doupdates) flag. The update list tail is used only by the GCP. When an update cycle is completed, the GCP returns the blocks in the update list to the free storage pool.

- First Display Control Block Pointer

The first Display Control Block pointer (FirstDCB) is an address in Mass Memory where the head of a list of Display Control Blocks resides.

- Size of Free Storage Block

This field gives the size in bytes of the contiguous block of free storage which has been allocated for the ACP's exclusive use for operations

such as sectioning and hidden line removal. A value of zero implies that no storage has been set aside for ACP use.

- Pointer to Free Storage Block

This field gives the pointer to the free storage block described above.

- ACP Status

This field is used as part of an intricate communications mechanism between the ACP and the GCP. When any solid modeling command is invoked via the `solid_rendering` function, this field is written to by the ACP to indicate either that solid modeling is in progress, that solid modeling has completed or that an error has been detected by the ACP. On the GCP side, this field is checked by the `solid_rendering` function to determine when the ACP has completed its part of a viewing operation. The ACP timeout function must also check this field (along with some other fields) to determine if the ACP has had a genuine timeout or not. If the ACP has detected an error while performing a viewing operation, an error number is written to this field to indicate the type of error. The error will be brought to the attention of the GCP, as either the `solid_rendering` function will see the error number in this field or the timeout function will see it (the ACP forces a timeout when an error has been detected.)

- Timeout Function

This field, used only by the GCP, is a pointer to a function instance which restarts the ACP and preserves any pending updates. The function is activated when the ACP requires 120 clock ticks or more to walk the display structures, which is usually when a recursive data structure exists.

- Display Control Block (DCB)

The Display Control Block (DCB) is a block of storage that is the highest level data structure for each user on the PS 390 system. There is one DCB for each user. The DCB is linked to the topmost nodes for all of the data structures associated with the user. The DCB is shown in Figure 2-7.

Next DCB
Pick Select List Head
Allocate Plotter
Plotter Select
First Set
Pick Window
Pick Active Flag
RB Picked x
RB Picked y
Pick ID
Rsvd for Future Enhancements
Rsvd for Future Enhancements
Rsvd for Future Enhancements
Rsvd for Future Enhancements
Rsvd for Future Enhancements
Rsvd for Future Enhancements
Rsvd for Future Enhancements
Rsvd for Future Enhancements
LPT Status
Pad for ACP
Light pen X position
Light pen Y position
Below Unknown to ACP
•
•
•

Figure 2-7a. DCB Block

TYPE

Int8 = -128..127
 PtrDCB = DCB;
 Ptrnamedentity = ^Namedentity;
 Ptralphablck = ^Alphablck;
 Dcb =

RECORD

Nxt: Ptrdcb;	{Next on list}
Picklh: Integer;	{Pick list head (really ptrnamedentity)}
Plotallocate: Bitmask;	{Bit 11=1 if plotter allocated}
Plotterselect: Bitmask;	{Bits 9..8 select plotter--used only by GCP}
Firstset: Ptralphablck;	{Pointer to alpha of set}
Pickwindow: Int16;	{Pick window size}
Pickact: Int16;	{Pick active flag}
RBPickx: Int16;	{Picked x from Refresh Buffer}

RBPicky: Int16;	{Picked y from Refresh Buffer}
RBPickID: Integer;	{Pick ID from Refresh Buffer}
F1: Integer;	{Reserved for future}
F2: Integer;	{Reserved for future}
F3: Integer;	{Reserved for future}
F4: Integer;	{Reserved for future}
F5: Integer;	{Reserved for future}
F6: Integer;	{Reserved for future}
F7: Integer;	{Reserved for future}
F8: Integer;	{Reserved for future}
LPTStat: Bitmask;	{Lightpen tip switch status and LPDelta,}
	{ LPAction bits}

{ Bit definitions for lptstat: }

{ Bit 15 - Blast	— blast found a pick }
{ Bit 14 - Tipdown	— tip switch depressed }
{ Bit 13 - CrossPick	— tracking cross found pick }
{ Bit 10 - Newxyon	— (x,y) position used }
{ Bit 9 - BlastOn	— screen blast enabled }
{ Bit 8 - CrossOn	— cross was enabled }
{ Bit 4 - ExdDelt	— x,y change exceeded delta }
{ Bit 3 - EdgeTrue	— tipsw edge true signal }
{ Bit 2 - EdgeFalse	— tipsw edge false signal }
{ Bit 1 - LPAction	— signal to sched LPIN }
{ Bit 0 - ACPAction	— signal to ACP to hold }

jdpad : Int16;	{Pad to 0 mod 4 for ACP}
LPXcen: Int16;	{Light pen X center, exp = 0}
LPYcen: Int16;	{Light pen Y center, exp = 0}

{ Below, not known to the ACP }

Menufcn: Ptrnamedentity;	{fcn waiting for menu pick}
Pickfcn: Ptrnamedentity;	{fcn waiting for pick}
Timeoutfcn: Ptrnamedentity;	{Fcn to handle ACP timeout}
LPfcn: Ptrnamedentity;	{fcn to handle light pen}
Usersfx: char;	{user suffix char, this DCB}
Menucopy: integer;	{copy of Menunum at Menu time}
F9: Integer;	{Reserved for future}
F10: Integer;	{Reserved for future}
F11: Int16;	{Reserved for future}
Actual_pblock: Ptrnamedentity;	{Actual block picked}
Pickindex: Int16;	{Pick index field of picked node}
Pickcount: Int16;	{Vector count of pick}
Pickblock: Ptralphablk;	{Picked block}
Dcbpad3: Int8;	
Picktype: Dattype;	{Dattype of picked node}

Pickerror: Int16;	{Pick error}
Pick3derror: Int16;	{3-D Pick error}
Pickexp: Int8;	{Exponent of coordinates}
Vectype: Int8;	{Vec type of picked vec}
Pickx: Integer;	
Picky: Integer;	
Pickz: Integer;	{x,y,z of picked vec}
Gamma: Integer;	{Curvet/Tsegments}
Transf_exp: Int16;	{3x4 Transformation exponent}
Transl_exp: Int16;	{1x4 Translation exponent}
PLS_conf_reg: Int16;	{PLS Configuration register}
Dcbpad4: Int16;	{Reserved for future}
Xprev: Integer;	{Transformed xprev}
Yprev: Integer;	{Transformed yprev}
Zprev: Integer;	{Transformed zprev}
Wprev: Integer;	{Transformed wprev}
Xcur: Integer;	{Transformed xcur}
Ycur: Integer;	{Transformed ycur}
Zcur: Integer;	{Transformed zcur}
Wcur: Integer;	{Transformed wcur}
Prev_Outcode: Int16;	{Previous Outcode}
Cur_Outcode: Int16;	{Current Outcode}
F12: Integer	{Reserved for future}
END; { of DCB }	

Figure 2-7b. Pascal Data Definition of DCB Block

A few details of the Data Control Block are explained below:

– Next DCB

The next DCB pointer (Nxt) is an address in Mass Memory of the next DCB on the DCB list. In multiple user environments, a DCB exists for each user. The ACP traverses the DCB blocks (and associated display data structures) until it encounters a next DCB pointer that is zero. For a single user, the next DCB pointer is always zero.

– Pick Select List Head

The ACP may encounter pick identifier nodes during traversals of the display data structures. The Pick Select List Head (Picklh) is a pointer to the Mass Memory location where the ACP last encountered a pick identifier node.

- Allocate Plotter

The PS 390 does not support a plotter. However some bits in this field may be used to communicate to the ACP.

- Plotter Select

The PS 390 does not support a plotter.

- First Set

First set is an indirect address in Mass Memory of the topmost node of all the data to be displayed for the workstation associated with this DCB. The node, or one of its descendants, is expected to contain members for each of the user display modes (i.e., Graphics, Terminal Emulator).

- Actual_pblock

After a pick has occurred, this field contains a pointer to the block whose vector or character was picked.

- Pick Index

The pick index is a 16-bit integer associated with a vector or character list, indicating which list contained the picked vector or character.

- Pick Count

The pick count is an integer indicating which vector or character was picked, respectively, in a vector or character list. This count is valid only when the pick select listhead is non-nil.

- Pick Block Pointer

After a pick has occurred, this field contains a pointer to the alpha block of the node whose vector or character was picked.

- Data Type

After a pick has occurred, this field contains a value indicating the data type of the vector or character list in which the pick occurred.

- Exponent, Vector Type, Picked x, Picked y, Picked z

These fields give the x, y, and z values of the point which was picked and provide information in the Vector Type(Vectype) field, concerning the state of the pick. For example, if an error occurs in determining the x, y, and z values, the ACP sets bit 7 of the Vectype field. Also, if no x, y, z values are to be returned (for example, for a character pick), bit 3 of the Vectype field is set.

- Curve_t

This field gives a count used to compute the (x,y,z) of the picked point as a fraction of the vector when the vector is part of a curve definition.

- T_segment number

This number indicates number of segments that the 3-D picking microcode has divided the original picked vector to find the 3-D coordinates.

- LPT status:

The PS 390 does not support a light pen.

- Light pen X position, Light pen Y position

The PS 390 does not support a light pen.

- Menu Function

Not currently used.

- Pick Function

The pick function (Pickfcn) is a pointer to the pick function instance which is activated when a pick occurs (when the pick listhead is non-nil). This field is known only to the GCP.

- Timeout Function

Timeout function (Timoutfcn) is a pointer to the user timeout function, which is activated when the ACP requires too much time to traverse the

display structures (usually indicating a recursive display structure). The user timeout function removes all user-created display structures from the display. Note that this field is known only to the GCP.

- Lightpen Function

The PS 390 does not support a light pen.

- User Suffix

The user suffix (Usersfx) is a character which identifies the user associated with this DCB. This field is known only to the GCP.

- Menucopy

Not currently used.

2.2.2.2 Set Node

A set node is a data structure that can contain a variable number of members, each of which is, in effect, a set node, operate node, or a data node. Each member of the set is processed independently from every other member. That is, the state of the ACP is guaranteed to either be saved before each member of the set is traversed and restored afterwards or to remain unchanged by the member of the set. As Figure 2-8. shows, a set node consists of:

- Set Node Indicator

The set node indicator is a value (=0) that identifies this Named Entity as a set.

- ACP Save State Pointer

The ACP save state pointer is an address in Mass Memory of the location where a block of storage has been allocated for the saving of the ACP state information. If this address is zero, the ACP state is not to be saved for this set.

- ACP Control Block Listhead

The ACP control block listhead is an address in Mass Memory where the first ACP control block associated with this set resides. An ACP control block exists for each member of the set.

- ACP Control Block Listtail

The ACP control block listtail is an address in Mass Memory where the last ACP control block associated with this set resides. This is always a dummy control block (the alpha pointer is always null). It is there because the ACP writes into its NEXT pointer to aid in structure traversal. The GCP is not allowed to remove this control block.

- GCP Control Block Listtail

This control block is the last real control block. The ACP does not know about this field. It is there so that the GCP can add control blocks to the end of the list of control blocks without walking through the entire list of control blocks.

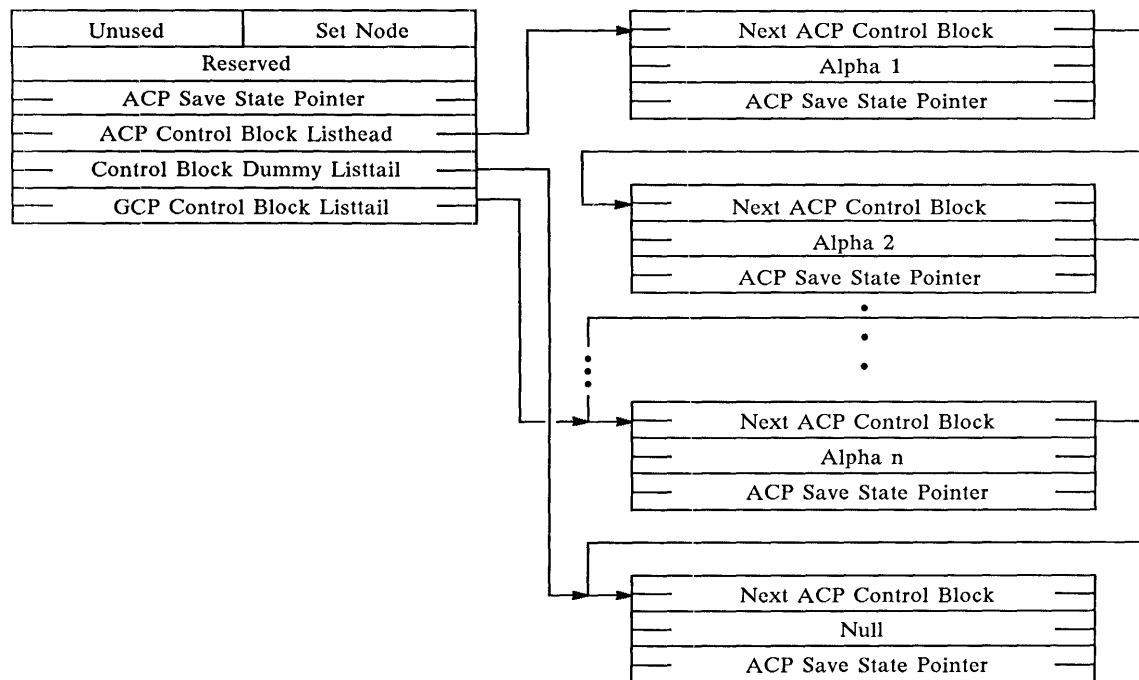


Figure 2-8a. Set Nodes

```

CONST
    Netyp          = 0;
TYPE
    Ptrsavstate    = ^Savstate;
    PtrACPcblk     = ^ACPcblk;
    Ptralphablk    = ^Alphablk;
    Int8           = -128..127;
    Int16          = -32768..32767;
    Namedentity    =
        RECORD
            Notused: Int8;
            CASE Typ: Netyp OF Setnode;
            Notused2: Int16;
            Ss: Ptrsavstate;
            Lh: PtrACPcblk;
            Lt: PtrACPcblk;
            Actlast: PtrACPcblk;
        END;
    ACPcblk        =
        RECORD
            Nxt: PtrACPcblk;
            Alpha: Ptralphablk;
            Ss: Ptrsavstate;
        END;

```

Figure 2-8b. Pascal Data Definition of Set Node

As figure 2-8 shows, each set consists of a variable number of ACP control blocks, linked together as a list that will be traversed by the ACP as the set is processed. An ACP control block exists for each member of a set. Each ACP control block consists of:

- Next ACP Control Block Pointer

The next ACP control block pointer (Nxt) is an address in Mass Memory of the next ACP control block to be processed after the display data structures associated with this ACP control block are traversed. (Refer to Alpha below.)

- Alpha

Alpha is the indirect address in Mass Memory of the display data structures to be traversed as this member of the set.

- ACP Save State Pointer

The ACP save state pointer (ACPstate) is an address in Mass Memory where the current ACP state resides. This state must be reloaded by the

ACP before traversing the descendent display data structures of this block. If the ACP save state pointer is zero, then the ACP state has remained unchanged from the last time it was loaded. For example, the first ACP control block of a set will always have an ACP state pointer of zero, since the ACP save state is the current state, by definition.

When the ACP encounters a set, the last ACP control block of that set (the one pointed to by the ACP control block listtail) is updated to point to the next ACP control block to be traversed when that set is completed. The ACP then traverses the set. When the last ACP control block of the list is traversed, the updated next-ACP-control-blockpointer then directs the ACP to the next ACP control block.

2.2.2.3 Operation Node

An operation node is a data structure that modifies the state of the ACP. As shown in Figure 2-9, an operation node consists of an integer that indicates this display data structure is an operation node (=1), an integer that specifies the particular type of operation node, the descendent alpha, and a variable number of fields required by the particular type of operation node. Because an operation node modifies the state of the ACP, the ACP state must be saved before traversing a member of a set whose descendants include an operation node, and be restored before traversing the next member of the set. For any operation node, bit 15 of the operate type is a Conditional bit. (A "C" in the left corner indicates this bit.) If this bit is set, and if bit 15 (the blink bit) in the Condition Mask of the ACP State is zero, then the associated operation node is not performed. In all other cases, the operation node is performed. In all cases, the son of the operation node is traversed. Each of the operation nodes is described in Appendix 9.10.

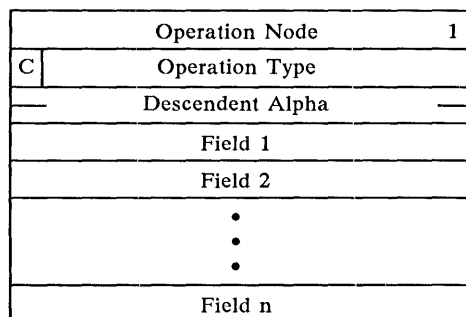


Figure 2-9. Operation Node

2.2.2.4 Data Node

A data node is the display structure primitive that causes data to be drawn by the ACP. A data node consists of an integer that indicates this display structure is a data node (=2), an 8-bit field that specifies the mode of vectors in the data node, an 8-bit integer that specifies the particular type of data node, a 32-bit integer which points to the next data node of identical data type, an integer (n) that specifies the number of vectors, polygons or characters in the data node, a 16-bit integer that specifies the pick index, and either vector data (including polygons) or character data. Vector data consists of the two- or three-dimensional vectors (preceded by polygon attribute information if polygons). Character data consists of an initial translation, spacing information, and the character string.

The general format of a data node is illustrated in Figure 2-10. Fields marked by asterisks are not used nor accessed by normal ASCII and GSR commands. The top bit in the second word of each of these formats (labeled "A") is a flag which, if clear, tells the display structure walker to process these fields. This bit is set by default, and there exists no command to clear it. Advanced user-written functions and programs using the physical read/write facility may however, use these fields and clear that flag. The format for each of the data nodes is shown in Appendix 9.10.

Data Node		2
A	Do_Dots	Data Type
Pointer to Next Data Node		
n		
Pick Index		
*	Line Texture	Traverse Count
*	Color	
	•	
	•	
	•	

Figure 2-10a. Data Node

```

TYPE
  Namedentity    = RECORD
    Not used: Int8;
    CASE Typ: Netype OF
      ACPdata: (Do_Dots: Boolean;
        Adson: Ptrnamedentity;
        Adnum: Int16;
        Adindex: Int16
        •
        •           {Data}
        •
      );
    END;

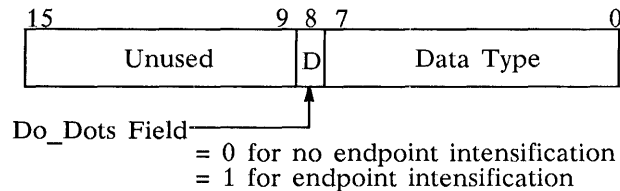
```

Figure 2-10b. Pascal Data Definition of Data Node

Mode, data type, pick index, and vector/character data are detailed further below.

- Do_Dots Field

The Do_Dots field of a data node consists of:



The Do_Dots field of a data node is a single bit that specifies how the vectors are to be drawn. When dot mode = 0, vectors are drawn normally. When dot mode = 1, each endpoint of the vector list is drawn as an intensified dot.

- Data Type

The data type field specifies the particular format of the data node. The ACP in the PS 390 accepts vectors of two formats: they are the 16-bit block-normalized format shown on the next page and a 32-bit block-normalized format that is the same in computation except that f_x , f_y , and f_z are 32-bits instead of 16-bits.

— Block-normalized data

Block-normalized data consist of 16-bit signed binary fractions that share a common 7-bit signed integer exponent and an explicit 8-bit intrinsic intensity for each block of vectors. For the vector:

$$\begin{aligned}x_1 &= 2^e * f_{x1}, & y_1 &= 2^e * f_{y1}, & z_1 &= 2^e * f_{z1}, & i &= i \\x_2 &= 2^e * f_{x2}, & y_2 &= 2^e * f_{y2}, & z_2 &= 2^e * f_{z2}, & i &= i \\&\vdots \\x_n &= 2^e * f_{xn}, & y_n &= 2^e * f_{yn}, & z_n &= 2^e * f_{zn}, & i &= i\end{aligned}$$

where e is the signed 8-bit integer exponent; the 16-bit significant digit fields fx , fy , and fz satisfy $-1 < f < 1$; the 7-bit intrinsic intensity field i satisfies $0 < i < 1$.

- Next Data Node Field

The next data node field contains a 32-bit pointer to the next data node of identical type (0 = nil pointer). This pointer allows vector lists to exist in non-contiguous blocks of memory and also allows one to group a set of character strings together (Label Block).

- Pick Index Field

The pick index field of a data node is reported with the vector count when a pick occurs, thus signifying the vector list in which the pick occurred. Although the number of vectors that may be contained in a data node is 65,535 (if n is treated as a 16-bit unsigned number), by convention, the maximum number of vectors that will be specified in a given data node is 2048, which is less than the maximum number of vectors that may be counted during pick processing. The software that creates data nodes will ensure that the index is correct for a given data node and that the reported index, together with the vector count, will allow one to correctly identify the actual vector that was picked.

- Vector or Character Data

- Vector Data

All vector data processed by the ACP are numbers of normalized, floating-point form, as $2^e * f$, where e is a signed-integer exponent and the significant digit field, f , satisfies $-1 < f < 1$. Rather than provide an exponent for each coordinate of a vector, the Display Processor associates a single exponent with each block of vectors. All vector data are two- or three-dimensional (i.e., (x,y) or (x,y,z)), with an implicit, homogenous coordinate equal to 1 (i.e., $(x,y,z,1)$). The dynamic range gained by explicit use of the homogenous coordinate has been provided by representing vector data in the normalized, floating-point form. In addition, polygon vectors have implicit closure; that is, there is an implied vector from the last point of the polygon to the first point. The ACP automatically displays this implied vector.

- Character Data

Character data consist of an initial translation to position the character string, spacing information to control the spacing between characters, and a string of characters. The initial translation consists of 16-bit, signed binary fractions for x , y , and z , with an implicit, homogeneous coordinate equal to 1 (i.e., $(x,y,z,1)$), and a shared 8-bit, signed integer exponent. Thus, the translation:

$$(x,y,z,1) \quad x = 2^e * fx, \quad y = 2^e * fy, \quad z = 2^e * fz,$$

where e is the signed, 8-bit integer exponent, and where the 16-bit significant digit fields fx , fy , and fz satisfy $-1 < f < 1$. The spacing information consists of a *delta x* and a *delta y*, each a 16-bit, signed binary fraction, sharing an implied exponent equal to zero. The *delta x* and *delta y* values determine the separation between characters in the x and y directions. They are given in the coordinate space of the characters themselves, satisfying the range:

$$-1 < = \text{delta } x, \text{ delta } y < 1.$$

For each character in a string of characters, the corresponding character stroke block is read from Mass Memory to provide the vectors which make up the individual character. The format of this character stroke block is described in Section 2.2.3.

2.2.3 Character Font Block

Each entry of the character font table is a 32-bit address in Mass Memory indicating where the associated character stroke block resides. Each character stroke block is an abbreviated Vec2s0 block, consisting of a count of the number of vectors followed by the Vec2s0 vectors. Note that this type of data block is used only in association with the character font block.

The character font block and associated character stroke blocks (with strokes in relative mode) are shown in Figure 11. The x and y components of each vector are 7-bit binary fractions that have an implied exponent of 0.

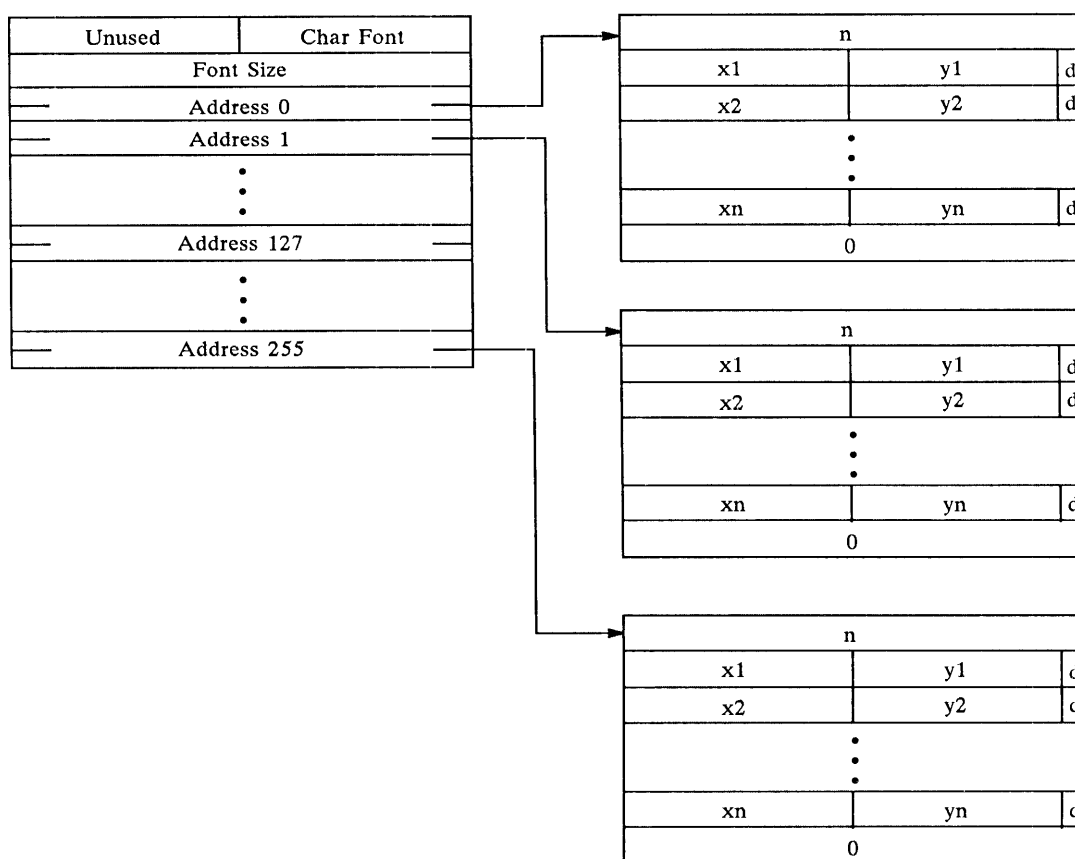


Figure 2-11a. Character Font Blocks


```

TYPE
  PtrVec2sblock  =  Integer;
  Ptrqdata       =  ^Qdata;
  Charfont       =
    RECORD
      Fontsize: Int 16;
      CASE Chars: Chsize OF
        C128: (Ccd: ARRAY[0..127] OF PtrVec2sblock);
        C256: (Cnd: ARRAY[0..255] OF PtrVec2sblock);
      END;
    Vec2sblock    =
      RECORD
        V2snum : Int16;
        V2s : ARRAY [1..1] of Vec2s0;
      END;
    Vec2s0        =
      RECORD
        x : Int8;
        y : Int8;
      END;

```

Figure 2-11b. Pascal Data Definition for Character Font Blocks

2.3 Commhead

The largest portion of global variables used by the PS 390 exists in a record called Commhead. Its format can be seen in the Appendix 9.9. This block, located at the low end of Mass Memory, contains pointers to the active and priority function lists, the various update lists, the standard character font, the hashtable, the command and function name dictionaries, and the initial Save State, as well as to other information.

2.4 Number Formats

The intrinsic data types utilized by the Graphics Control Program are:

- Int8:
an 8-bit, two's complement number in the range of -128 to +127.
- Int16:
a 16-bit, two's complement number in the range of -32768 to +32767.

- Integer:
a 32-bit, two's complement number in the range of -2147483648 to +2147483647.
- Double:
a 48-bit precision, floating-point number consisting of a 16-bit exponent (or characteristic) and a 32-bit fraction (or mantissa).

2.5 Hash Table

The hash table is an array [-1..n] of pointers to forward and backward linked lists of alpha blocks. The value of n is based on the amount of mass memory available when the system boots. This value is contained in the Commhead.

Naming a node causes the name to be entered into the hash table with a pointer to the Alpha block associated with that name in PS 390 Mass Memory. The -1 entry in the table is for forgotten Fcninstances since their connections must be found when doing an INIT CONN command.

Any time a name is used in the system to reference a structure the name is processed by a hash algorithm to produce a hash index. This number is the index into the hash table. The Alpha pointer at that index is checked to see if the name equals the requested name. If not, the dictionary forward pointer of the Alpha is then checked. This process of checking continues until the name is found or the linked list ends.

Section AP3

Internal Processing

3.1 Structure Creation

There are four main steps in the creation of structures in the PS 390.

1. Alpha Lookup
2. Named Entity Creation
3. GCP Datum Pointer Set
4. Alpha Update

3.1.1 Alpha Lookup

A lookup of the name specified is done which returns with a pointer to the Alpha block of the specified name. If the name is not already in the dictionary, it is added during the lookup.

3.1.2 Named Entity Creation

After the Alpha is established, the actual Named Entity structure is created. If the Named Entity is a function instance, there is a check of the function table to determine if it is a valid function. If it is valid, a check is made of the number of inputs and outputs to allocate in the function block. At this point the initialization code is executed and the state of the function is set to ACT ON UPDATE. If the Named Entity is a display structure, a Named Entity block of the proper type is created and data is put into the structure.

3.1.3 GCP Datum Pointer Setup

A pointer to the Named Entity is placed in the GCPdatum of the Alpha block.

3.1.4 Alpha Update

Pointers to the Alpha block and the Named Entity are placed on the ACP update list that causes the change to take place at the end of the current structure traversal.

3.2 Update Process

The GCP creates and manipulates the display data structures in mass memory (and initiates the display defined by these data structures). The ACP then accesses the data structures in Mass Memory, traverses the structures, and transforms the data to be displayed. A rigid update process is followed when modifying display data structure. This process ensures that the GCP won't corrupt a data structure being traversed by the ACP and that changes in the display data structures will be synchronized. The update process entails:

1. A function produces a private list of changes through calls to Lgaupdate and OLbaddtoet. This creates a list head and tail pointer containing the private list of updates. Each call to Lgaupdate or OLbaddtoet adds to the list until a call to Announceupdate is made. The list tail is used in calls to FetchBlock, FetchAdnum and nFetchCopy to allow the searching of the private list for matching Named Entities.
2. The function hands the private list to the update formatter. This list is made available by a call to Announceupdate. This call gives the functions private list to the update formatter to be added to the current global list of updates to the ACP.
3. The ACP makes the changes specified in the global list at the end of a refresh cycle. The ACP processes the list of changes and performs all updates.
4. The GCP post-processes the update blocks. After the ACP finishes the update list, the blocks of data are returned as available storage.

Because the display data structures are constantly being traversed for display by the ACP, whenever a change is required to a node the GCP "usually" makes a copy of the node that is to be changed, changes the elements to be changed, and then simply causes the pointer to the node in the alpha block to be changed by the ACP after it has finished refreshing. This occurs whenever a matrix is changed, a new PS 300 display enabled, etc. In essence then, most changes to the display data structures cause the change to be "double buffered" until the change can take place. Such changes are made by the ACP at the end of each refresh cycle.

The GCP creates an update block whenever display data structures require updating by the ACP. Basically, an update block indicates what changes

need to be made and where these changes are located in Mass Memory. The update block also specifies what TYPE of update is to be processed. The ACP can process two types of update blocks:

- Alpha update
- Value update

3.2.1 Alpha Update

An alpha update is performed whenever an alpha block is to refer to a new Named Entity block. Except for minor changes in data nodes, most updates in the standard runtime system are alpha updates. For example, all operation nodes are updated by creating a new operation node with the correct contents, then performing an alpha update.

3.2.2 Value Update

A value update is performed whenever a value is to be updated in an already existing data structure, i.e., whenever small portions of a data structure (e.g., character(s) or coordinates of a point) need to be updated in place without copying the entire data structure.

3.2.3 ACPProof

The updates process is not universally followed. A technique, called ACPproof, was established for special cases when direct modification by the GCP of the data structure (without going through the normal update process) is needed. If the GCP makes a change to a data structure through the normal update process, a Named Entity can NEVER be expected to be in the same location in Mass Memory. However, conversely, garbage collection is never done on existent Named Entities by the PS 390. So once a node has been created, if the node is NEVER referenced by a standard PS 390 function network or externally from the host, a Named Entity can ALWAYS be expected to be in the same location in Mass Memory. This knowledge can sometimes be used in a user-written function or by physical I/O programmers.

For example, values in the node can be changed directly. However, if the node is currently being displayed no guarantee can be given that the ACP will not traverse the node during the small, but finite, time frame during which the data may be changed. This may result in an improper picture

being displayed (new x value, old y value; or a matrix with mixed values—part new, part old) but should never cause the ACP to traverse the display data structures improperly, **As Long As No Pointers Are Changed**. Any pointers which are to be directly changed in the data structure by the GCP should use ACPproof. ACPproof uses a convention with the ACP that if the top half of a pointer is zero, the pointer should be treated as nil. (ACPproof works by zeroing the upper half of a pointer, changing the lower half, and then changing the upper half. In this way the ACP either gets the old pointer, gets the ACPproofed pointer with a zero upper word, or gets the new pointer.) Not all pointers can be modified in this way. The ones which can be used with ACPproof are:

- Nxt in the DCB
- Firstset in the DCB
- Nxt in Acpcblk
- Alpha in Acpcblk
- Ason in Acpoper
- Adson in Acpdata

3.2.4 Use of RAWBLOCK

The RAWBLOCK command is used to allocate memory that can be directly managed by a User-Written Function or by the physical I/O capabilities of the Parallel Interface.

The command:

name := RAWBLOCK i;

carves a contiguous block of memory such that there are “i” bytes available for use. Since this has to be a display data structure and one contiguous memory block, it is structured so that it appears as shown in Figure 3-1.

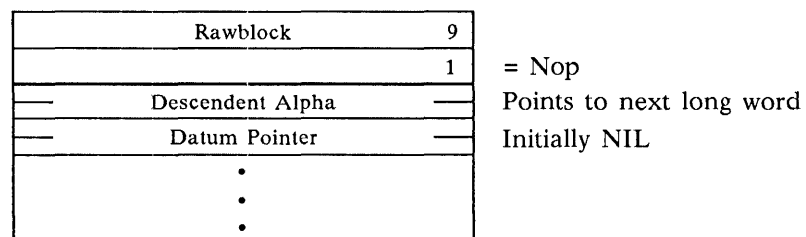


Figure 3-1. Rawblock

This block looks like an operation node to the ACP. The descendant alpha pointer points to the next long word in the block. What the ACP expects in this word is the datum pointer of the alpha block. This is initially NIL to make the ACP think that the alpha doesn't have any data associated with it yet. To use this block, the parallel interface or a user-written function fills in the appropriate structure following the datum pointer. When this is complete, it changes the datum pointer to point to the beginning of the data using ACPproof.

More than one data structure at a time can exist in a RAWBLOCK. It is up to the user to manage all data and pointers in a RAWBLOCK. A RAWBLOCK may be displayed or deleted like any other named data structure in the PS 390 (e.g., DISPLAY "name"; or DELETE "name";).

3.3 Function Operation

The PS 390 functions are instances of "generic" functions which exist in the PS 390 runtime system. Generally, a generic function is a Pascal procedure which performs one or more operations by (a) accepting input, (b) processing input, and (c) sending output. (Note that occasionally the function code is written in assembly language but is called as a Pascal procedure.) The user, or less commonly the runtime software itself, creates a PS 390 function by "instantiating" a generic function. There may be many "instances" of the same generic function. Each instance has its own user- or system-defined name, as well as its own input queues and output connections. The user connects these function instances into a network. Thus, each PS 390 function generally has inputs coming from other functions and outputs going to other functions or data structures. All information regarding the instantiation of a function, as well as all information regarding connections between instances, is kept in a "function instance block" and its associated substructures.

When a function instance is created, it is assigned a default priority for execution (most default to 8). The Scheduler uses priority numbers to determine which of the PS 390 functions awaiting execution will be executed next. It executes a function by placing a pointer to the function instance block and calling the Pascal-callable procedure for the generic type of the function instance.

A function instance cannot be executed until all of its essential inputs have arrived. Inputs exist on (and outputs are sent to) input queues. Once the function has processed the inputs, any output values are sent to destinations listed as outputs of that instance. Some functions may wait on an I/O device rather than on an input queue. In this case, the I/O device interrupt routine activates the function at the proper time. The following sections detail the creation, manipulation, scheduling, and execution of function networks.

3.3.1 Scheduler

Although it is not the major portion of code, the Scheduler is the driving force behind the Graphics Control Program. The scheduling loop is the process by which the Scheduler executes activated functions. Executing a function means to place the address of the function instance block in a global variable and call the generic Pascal procedure.

The Scheduler is designed to avoid two major sources of lost time in software-scheduled systems: task context switching and the scheduling decision itself. Time can be lost in context switching, i.e., when a program is interrupted, because the complete processor state (including any memory mapping state) must be saved and the state of another task must be put in its place.

The Scheduler avoids this loss in processing time by assuming that all scheduled functions will run to completion once execution begins. This assumption is based on the constraint that all functions complete in a reasonable amount of time, i.e., less than two milliseconds. If an operation could take much longer, the function must place all data on a private queue and reschedule itself for execution. Note, however, that because this saving, restoring, and rescheduling process is so time-consuming, some functions are allowed to continue running.

Because of the large amount of time to schedule a function instance, most functions are allowed to process more than one set of inputs per wakeup if a global Boolean variable `KEEPPGOING` is true. This variable is set to be true by the scheduler prior to executing a function. However, any time the clock interrupt routine sees that a function waiting on the clock is ready to run, it can potentially set `KEEPPGOING` to be false. Once `KEEPPGOING` is false, the currently running function must give up control prior to processing the next set of inputs.

Time would also be lost in scheduling decisions if the Scheduler scanned all potentially active tasks to determine which were truly active. Therefore, the Scheduler loop scans only those functions that are part of an active list. When a function instance is created, it is assigned to one of 16 possible priority levels (0–15). The priority level is used by the Scheduler to determine when the function instance will execute. Unless mass memory is nearly full, priority is such that the smaller the priority number, the earlier the function instance is executed. Most functions operate at priority level 8. If mass memory is nearly full, the Scheduler executes only functions which do not require additional mass memory.

At each execution, the Scheduler empties the Active List (the list of functions to be executed) into a set of separate lists (according to priority). It then executes the first function on the highest priority list. After the function has been executed, control returns to the Scheduler and the process is repeated. Note that user commands do exist to change a function's priority level. However, if a function is already on an active list when its priority is changed, the function's scheduling position does not change until subsequent activation.

3.3.2 Function Activation

Function instances are “activated” when they are placed on the Active List. The Scheduler then processes the Active List and executes the functions. Once a function has been instanced, it must have received all of the inputs needed for execution in order to be activated. During function instance creation, the function instances are initialized if necessary. This generally means that default values are placed on some input queues, and the private data message is created and initialized. Function instances are first activated following function instance creation. The first time it is executed a function will cause itself to do one of the following:

- Be reactivated.
- Wait for an input on one or more input queues.

A function waits for input on its queues by setting the Numnonnull counter in its record block to indicate the number of necessary input queues which still remain empty. Whenever another function sends data to an input queue, if that queue does not already contain data (hence one more necessary queue now has data), the numnonnull counter is decremented. When this counter is decremented to zero,

the function is placed on the active list by the procedure doing the message send.

- Wait for the clock or an I/O event.

If a function instance has an output designation and is ready to send a message to another function, it does so by:

- Obtaining a Qdata block from free storage and creating the message.
- Sending the message to the destination specified in the function block's outdesignator. Utility procedures exist to do this.

3.3.3 Function Status

At any given time during its instancing or activation, a function exists in one of several possible states. Those states are:

1. Actonupdate

When the function is instanced. Once it has been tied to its name by the ACP, it takes on Active status.

2. IO_wait

When the function is waiting for input from an I/O device or waiting to be activated by the clock.

3. Msg_wait

When the function is waiting for input on one or more input queues.

4. Active

When the function is on the Active List (waiting to be executed).

5. Running

When the function is being executed by the Scheduler.

6. Self_destruct

When the function is to be destroyed, rather than executed, the next time it is scheduled.

Note that before a function is executed, the Scheduler changes its status to Running. A fatal system error occurs if control returns to the Scheduler after execution and the status is still Running. Thus, a function must cause its state to change during execution by either (a) waiting on a device or queue, or (b) activating itself, or (c) setting its status to Self-destruct and then activating itself.

A function cannot wait on more than one item (queue, clock, or I/O device). Thus, a system error also results if a function waits on a device, clock or input queue and then attempts to wait again without first changing its own state.

3.3.4 Function Code Format

The Pascal-callable procedure defining the generic function generally follows a rigid framework. It usually has a single parameter, which is a pointer to the function instance block of the particular function instance. A typical function procedure includes instructions to do the following:

1. Check input queue(s) for new data. If there is a complete set, go to step 2. If not, set status to wait for data to be sent to empty queues and return to the Scheduler.
2. Take one set of input data from the input queues (buffer).
3. Use that data to modify the private data, display structures, and/or generate output messages as needed.
4. Send any output messages to all destinations referred to in the function instance block.
5. Check the input queues for another sufficient set of data. If it exists, and the global flag Keepgoing is still set, then proceed to Step 2. If Keepgoing is False, then queue self on the active function list. If a sufficient set does not exist, set status to wait for data to be sent to empty queues and return to scheduler.

Section AP4

Physical I/O Programming

The PS 390 is designed primarily to meet the needs of those customers who require that the dynamics of picture display be handled on the local PS 390 level, rather than be tightly coupled to a host machine. This is possible through the means of function networks, which offer a selection of local actions, driven by peripheral devices such as the dials or data tablet, or simply by internally-generated time pulses. This removes most of the frame-by-frame load from the host, freeing it for other work. However, there remain many applications where the host itself may be required to closely direct — perhaps even on a per-frame basis — the dynamics of the displayed picture. The difficulty in meeting this requirement has to do not only with the overall software speed, but with an inherent limitation of the hardware; namely the narrow bandwidth of the communication channel from the host to the PS 390. Because it was intended that frame-by-frame dynamics be handled on a local level, the asynchronous interface was not designed for host-driven dynamic communication. To circumvent this problem, and to provide for efficient host direction of dynamic operations, the system function F:USERUPD was provided.

4.1 The F:USERUPD Function

This function permits a variety of dynamic transformations to be sent from the host each frame, directly effecting changes to the displayed picture on a per-frame basis. Only the arguments for these transformations are sent, with the matrices and vectors being generated by the F:USERUPD function, thus greatly decreasing the bandwidth requirements of the communication line. For example, the dynamic arguments for a picture with 25 to 30 degrees of freedom may be sent at a 10-hertz update rate over a 9600-baud, asynchronous line. The F:USERUPD function, however, has some limitations both in system response speeds, and in flexibility. The ability, for instance, to turn a portion of the picture on or off using level-of-detail or conditional bits is not available with this function. Refer to *Appendix 9.12*, for more complete details of the USERUPDATE function.

4.2 The Parallel Interface

In order to help meet the need for more closely-coupled host control of the PS 390, a 16-bit-wide parallel interface, operating through the General Purpose Interface Option (GPIO) was developed. This interface gives VAX users a much higher data transmission rate, on the order of 0.5 megabytes per second (in practice the rate is somewhat less, and depends on cable length). This is still many times faster than the 56 kilobaud runner-up.

The effective speed of this interface is so great, that it outstrips the ability of both VAX and PS 390 software to keep up with it. The Graphics Support Routines (GSR's), for example, must issue a separate System I/O Request (QIO) for each message to be sent to the PS 390. These requests are queued up to await the attention of the parallel interface device driver, and the more queued-up QIOs, the slower the response. In addition, on the PS 390 side, the incoming messages must each go through the command interpreter, generating new data nodes and involving other overhead having to do with the data structure.

4.3 Physical I/O

To fully take advantage of the speed of the interface, and to eliminate as much of the node juggling and other overhead as possible, the parallel interface protocol (and especially the GPIO microcode) includes, in addition to the standard communication commands, another set of commands collectively referred to as physical I/O commands. These commands permit the host to directly access the internal contents of any node (or other PS 390 structure, for that matter), and modify those contents at machine speeds, without any node swapping, pointer juggling, memory management, or command interpretation.

This direct access is possible because one of the physical I/O commands allows you to give the ASCII name to the PS 390 of any node, and receive back the physical memory location of that node. Once this address is known, and you know the internal structure of the node, you may use the Write Physical command to directly modify those contents to suit your needs.

In addition, physical I/O includes the capability of scatter-writing from a single buffer into many non-contiguous blocks of PS 390 memory, thus allowing a single host QIO to effect modification of all the dynamic updates for an entire frame.

4.3.1 Physical I/O Constraints

Because of the ability of physical I/O to circumvent the normal protocol of node and structure building by addressing any desired PS 390 mass memory location, you must use considerable caution in selecting the memory areas you choose to modify. You must consider the following rules to avoid crashing the PS 390 or worse by causing a bug which may appear later. These rules are:

1. Only the contents of ACP data structures (i.e. nodes) can be modified. Modification of any other areas of memory is not allowed.
2. Only the DATA portions of the nodes can be modified. The STRUCTURE elements (in particular, the first four 16-bit words of each node) must never be modified.
3. Because the system modifies a node by making a copy of it in another place, modifying the copy, and then changing pointers, you must NEVER modify a node which can be modified by another source (for example, one that is referenced by a function network).
4. Note that the graphics display processor (the ACP) is also traversing the data structure at the same time your buffer is being written into it via physical I/O. A “double buffering” scheme must be implemented to avoid the chance of the ACP trying to access YOUR node while YOU are writing into it. Refer to 4.4 below.

4.3.2 Physical I/O Operations

There are four operations supported by the GPIO microcode to perform the physical I/O functions. These operations provide for (1) doing a name lookup, (2) doing a physical read of PS 390 mass memory, (3) doing a physical write to PS 390 mass memory, and (4) doing a synchronous physical write to PS 390 mass memory. The interface-specific commands and options for these operations are described in detail in the *Customer Installation and User Manual* for the appropriate interface. This section will describe the general data formats used in the physical I/O operations.

- The first of these is the lookup format. The lookup requires a name, consisting of a string of characters and a 32-bit integer variable where the address of the named entity can be returned by the GPIO. Only one name can be looked up per QIO. If there is no Alpha for the specified name, a null is returned for the address.

- The physical read requires a special list of addresses to read from PS 390 mass memory. The addresses acquired through multiple lookupname calls are assembled into the addrlist. The format of this list is shown in Figure 4-1.

Reserved	(used in Ethernet data transmissions)
Number of blocks to read	n <=255
Block #1 source	LS, MS Address of data read
Block #1 word count	Number of 16-bit words read
Block #2 source address	
Block #2 word count	
•	
•	
•	
Block n source address	
Block n word count	

Figure 4-1. Format of Physical Read Address List

When the physical read completes, it returns a list of addresses and data in the format shown in Figure 4-2.

Reserved	
Number of blocks to read	n <=255
Block #1 source address	LS, MS Address of data read
Block #1 word count	Number of 16-bit words read
Block #1 first data word	Data returned after read
•	
•	
Block #1 last data word	
Block #2 source address	
Block #2 word count	
Block #2 first data word	
•	
•	
Block #2 last data word	
Block n source address	
Block n word count	
Block n first data word	
•	
•	
Block n last data word	

Figure 4-2. Format of Data From PS 390 in Physical Read

- The physical write transfers a list of data to the PS 390 memory. The format for this list is shown in Figure 4-3. Note that the format is exactly the same as the data returned on a read. This allows you to do a physical read on a set of named entities, modify the data in the read list (do not modify the addresses), and write back the same list to the PS 390.
- The physical I/O write synchronous operation ensures that each buffer gets at least one refresh before allowing the next write operation. It is possible to specify that the physical write operation be synchronized with the ACP clock. The format for the assembled data block in synchronous physical write is identical to physical write, shown below.

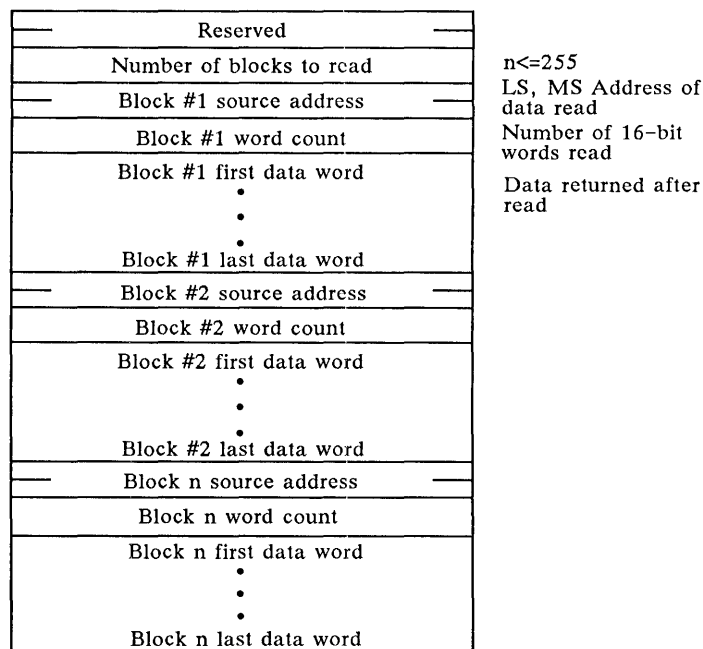


Figure 4-3. Format of Data to PS 390 in Physical Write

4.4 Advanced Physical I/O Programming

The Physical I/O process can produce distorted pictures when it is updating display structures at the same time the display processor is traversing them. To avoid this “single buffer” occurrence, these display structures can be “double buffered.” This is done by creating two copies of the named entities to be updated with different names (e.g. Data1 and Data2). The data structures can then be alternately updated and displayed using either the IF LEVEL_OF_DETAIL or IF CONDITIONAL_BIT commands such as:

```
TOP:=BEGIN_STRUCTURE
LOD:=SET LEVEL_OF_DETAIL TO 1;
      IF LEVEL = 1 THEN Data1;
      IF LEVEL = 2 THEN Data2;
END_STRUCTURE;
```

or

```
TOP:=BEGIN_STRUCTURE
CB:=SET CONDITIONAL_BIT 1 ON;
      IF BIT 1 ON THEN Data1;
      IF BIT 1 OFF THEN Data2;
END_STRUCTURE;
```

These commands are used in conjunction with a node higher in the structured display file that either sets the level of detail (SET LEVEL) or sets the conditional bit (SET BIT).

The node that performs the SET BIT and SET LEVEL operation is the Change Bits operation node. This operation node is also used to set displays, set character orientation, set contrast, set CSM, set depth clipping, set plotter, set rate external, set blinking, and set line texture. The format and a more detailed description of this node is contained in *Section 9.10* of this document.

The SET LEVEL or SET BIT nodes can be updated using the physical I/O to “swap buffers.” Placing the update of the SET LEVEL or SET BIT structure last in the physical write list will ensure that the data are all correct before the buffers are swapped.

Section AP5

User-Written Functions Tutorial

This section illustrates how to construct and use a simple user-written function (UWF). A user-written function is a Pascal procedure that will accept input data, process the data, and output the resulting data. User-written functions can be designed to perform operations not supplied by standard PS 390 functions and also to collapse large function networks into a single function.

5.1 Introduction to User-Written Functions

The User-Written Function facility is provided to allow you to expand and enhance the usefulness of your system by writing functions of your own design. User-written functions can be written to create new functions that perform operations not provided by intrinsic PS 390 functions.

User-written functions may also be written to perform tasks that would require a large network of intrinsic functions to accomplish. For example, numerical calculations are usually easier to perform inside a single user-written function than within a function network.

Substituting a single user-written function for a large network of intrinsic functions can be beneficial in two ways. First, programming a few functions in Pascal may be easier than programming a complicated function network. Second, due to the overhead incurred by scheduling each function in a large network, a single user-written function will usually take less execution time. When collapsing a large function network into a single user-written function, there are several considerations:

- User-written functions execute somewhat more slowly than intrinsic functions. Therefore, collapsing a network consisting of just a few functions into a user-written function may not result in any improvement in performance.

- It is usually not possible to replace an entire function network with a single user-written function. User-written functions tend to be more useful for performing specific tasks within the context of a larger network. Before writing a user-written function, you should be sure that the function has a well-defined purpose and a definite set of inputs and outputs.

5.1.1 Requirements

No separate hardware is needed to write your own functions. You must, however, be able to communicate with your host system. If your PS 390 is not equipped with terminal emulator capabilities, you will need a separate terminal to communicate with your host and access host-resident utilities.

To write your own functions, you will need the Motorola 68000 cross-software (compiler, assembler and linker). Before using the tutorial section of this manual, the Motorola software must be resident in your host system and available for use.

The Motorola software may be purchased and licensed through E&S or from Motorola directly. The software available through E&S has been modified to run in DEC VAX/VMS and VAX/UNIX environments; the software purchased from Motorola supports IBM (specifically MVS/TSO) environments. Further information on purchasing the software and the license can be obtained from your E&S Account Executive.

You will also need two E&S-provided files, USERLINK.RO and USERSTRUC.PAS. These files are provided on magnetic tape and must be loaded on your host system before you can use the tutorial section of this manual.

Command files that are provided for the tutorial section of this manual were written for the E&S-modified Motorola cross-software. Modifications to the files may be necessary if any other cross-software is used.

These files are only provided for DEC VAX/VMS or DEC VAX/UNIX hosts. Users in an IBM environment should consult sections 9.3 and 9.4 for instructions and files that illustrate the use of the cross-software.

5.1.2 Objectives

In this section, you will learn:

- The steps for constructing a sample function whose template should be used in writing user-written functions.
- How to write your own function.
- How to compile, link, and name the function.
- How to transfer the function to the PS 390.
- The restrictions on instantiating the function.
- How to use basic debugging techniques.

5.1.3 Prerequisites

Before beginning this section, you should be familiar with the Pascal programming language, the use of PS 390 standard functions, and the downloading utilities on your host system. You should also make sure that your host system has the prerequisite Motorola compiler and linker software and that you have access to it. (IBM users should be familiar with the instructions in Section 9.3 of this manual.) You should have a PS 300 console and keyboard with terminal emulator capabilities, or a separate terminal that can communicate with your host system. Recommended books to have on hand that may be referred to are:

- *PS 390 Document Set*
- *Motorola Pascal User's Guide*
- Host-system utilities manual

5.2 Constructing a Simple Function

The first step in creating your own user-written function is writing the Pascal procedure that will later be compiled and transferred to the PS 390. The Pascal procedure must contain:

- Calls to internal PS 390 functions and routines that allow your function to be scheduled and run.
- All the code necessary for your function to read data, process data, and write data.

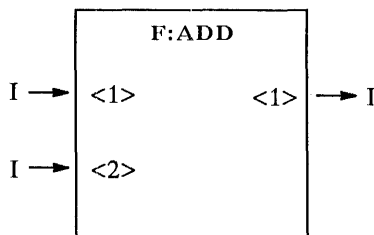
Like all standard PS 390 functions, the function you write must wait until it has an input value on all queues, perform its computations, and then output

the new values. All PS 390 functions must perform the same general series of actions when they are activated. These are:

1. Fetch messages from the input queues. The function must have a message available on every queue before it can run.
2. Make sure the messages received as inputs are of the appropriate type. If not, signal an error.
3. Perform whatever calculations are necessary.
4. Send output messages.
5. "Clean up" the input queues and see if the function can be run again immediately. If so, go to Step 1.

The following diagram and text illustrate a simplified version of the PS 390 function F:ADD and the Pascal procedure that supports it.

Function



Description

F:ADD accepts integers as inputs and produces an output that is the sum of those integers.

5.2.1 Example

Note that the example contains comments. Some of these comments will be referred to in following portions of text. Refer to the Section 8 of this manual for descriptions of the utility routines. This function, like all of the other examples included in this manual, was developed under DEC VAX/VMS. If you are using UNIX or an IBM system for developing your user-written functions, you may have to make minor changes in the examples. Refer to the section 9 for details on modifications and instructions on the use of the cross-software on your system.

```

SUBPROGRAM UWFadd;
{$F=USERSTRUC.PAS}
PROCEDURE GenFunction; {procedure body must always
                        {be named GenFunction}

VAR
  inputs   : PtrUWFInQarray; {pointer to data types in Qarray}
  outmsg   : PtrUWFInQarray; {pointer to data types in Qarray}
  i        : Integer;
  {-----}
  { Main body of UWF }
  { calls utility routine to check inputs for data, }
  { checks inputs for valid data type, }
  { returns error message if data are not valid. }
  {-----}
BEGIN {GenFunction}
  inputs := CkInputs (1, 2); {check for data on range of queues}
  WHILE inputs <> NIL DO BEGIN
    IF inputs↑[1]↑.qtyp <> Qinteger THEN {check for valid data type}
      QillMessage (1) {error message if data are invalid}
    ELSE IF inputs↑[2]↑.qtyp <> Qinteger THEN {check for valid data}
      {type }
      QillMessage (2) {error message if data are invalid}
    ELSE BEGIN
      {-----}
      {Allocate a new Qinteger to hold the output message. }
      {Then add the integers and send the sum from output <1>}.}
      {-----}
      outmsg := Newqinteger; {allocate memory block for output message}
      outmsg↑.i := Inputs↑[1]↑.i + Inputs↑[2]↑.i; {put sum in output}
      {message }
      Sendmsg(outmsg,1); {send message to output <1>}
    END;
    {-----}
    {Call utility routine to flush queues and see if there is}
    {enough time to process more data; call CkInputs to see }
    {if there is data on all queues. }
    {-----}
    IF CleanInputs THEN {flush input queues and see if there is enough}
      {time to process new data }
      inputs := CkInputs(1,2) {check for data on queue <1> thru <2>}
    ELSE
      inputs := NIL {get out of WHILE loop}
    END;
  END.

```

From the example, you can see that the first requirement of the Pascal procedure is that it checks for inputs on both queues:

```
inputs:=CkInputs(1,2); {Check for inputs on queues 1 through 2}
```

This will hold true for any user-written function; nothing can happen until the function has data on all input queues. The queues are checked for input by CkInputs, a utility function that accepts the range of the queues as parameters; for example, if the function had six queues, that line of code would read:

```
inputs:=CkInputs(1,6); {Check for inputs on queues 1 through 6}
```

CkInputs has a pointer to each of the input queues specified in the inclusive range and stores them. When CkInputs is called, if there are data on all of the queues, it will return with a pointer to the array. CkInputs will return a NIL if there are queues in the range that do not have input. Notice that if NIL is returned, the F:ADD function will exit. In F:ADD, if there are data on both queues, the program can proceed. The procedure next checks to see if the data on input <1> are the specified data type:

```
IF inputs↑[1]↑.Qtyp <> Qinteger THEN {input on <1> must be integer}
```

In this simplified version of F:ADD, the only acceptable data type is an integer. If the data type is not an integer, an error message is triggered, and the function cannot run. The utility routine, Qillmessage, would print out the message:

```
Message which function cannot handle.
```

signifying that the data on the queue were not of the specified type.

Input <2> is then checked for an acceptable data type and the same process is repeated. In any function, all input queues must be checked to see if the data on the queues are the specified data type. Further, the specified data type must be one of the QData types defined in USERSTRUC.PAS. (This list is provided in the reference section of this document.)

If both inputs have data, and data are in the specified range, the function can run. In this case, the function processes the data by adding the two integers together to produce a sum. Note that before the integers are added, a memory block is allocated for the output message with the statement:

```
outmsg := NewQinteger;
```

Memory must always be allocated for the processed data that will be placed on the output queue(s) of the function. Again, the outmsg must be one of

the QData types defined in USERSTRUC.PAS. After the memory is allocated and the integers are added together, the sum is then sent as an outmsg to output <1> of the function:

```
outmsg↑.i := inputs↑[1]↑.i + inputs↑[2]↑.i; {put sum in output}
                                         {message      }
Sendmsg(outmsg,1); {send message to output <1>}
```

Finally, the program flushes the input queues by calling the CleanInputs function, which also checks to see if there is more time available to process more incoming data:

```
IF CleanInputs THEN {clean up input queues and see if there is}
                   {enough time to process new data           }
```

The utility function CleanInputs should be called after the input data have been processed and the outputs have been sent. This function “cleans up” the input queues and determines whether there is enough time for the function to run again immediately. CleanInputs will return a FALSE if the function has been running for more than 2 centiseconds. Then, the inputs are again checked for data by calling CkInputs:

```
inputs := CkInputs(1,2) {check for data on queue <1> thru <2>}
```

If inputs = NIL, there is no more data on the input queues and the function exits. This is a very simple example and demonstrates the basic principle behind writing the Pascal procedures that will be used as functions in the PS 390 system.

The utility routines (or functions) that this program calls, CkInputs and CleanInputs, are just two of the utility subprograms that will be used in writing your own functions. These routines and functions allow for scheduling and communication between functions. A complete list of the utility subprograms are in Section 8 of this manual. Most of them will be described and demonstrated in this section and in the Advanced Ideas section. The utility routines and functions are declared in USERSTRUC.PAS, along with the QData types already mentioned. USERSTRUC.PAS must be compiled along with your user-written function by using the inclusion:

```
{ $F=USERSTRUC.PAS }
```

immediately after the name of your program. (See example.) It is important to remember that the rules that apply to standard PS 390 functions also apply to any functions that you will write. The first important rule to re-

member is that there must be a message available on all input queues before the function will be activated.

5.2.2 About Messages and Queues

The input and output messages received by the function must belong to one of the QData types declared in USERSTRUC.PAS. These message types include all of the types used by intrinsic PS 390 functions: integer, string, Boolean, real, vector, and matrix. In addition, it is possible for user-written functions to define additional message types; this will be discussed in more detail later on.

By default, all of the input queues for a user-written function are initially active queues, although you may use the SETUP CNESS command to establish some of the queues as constant queues. Within the code for the body of the user-written function, however, both constant and active queues are treated identically.

For your function to work properly, you must be careful to use messages correctly. Improper use of messages is by far the most common source of problems with user-written functions. Failure to observe the rules for proper use of messages will, at the very least, cause your function to behave unpredictably, and may cause the PS 390 to crash.

Messages used by a user-written function are of two types: those that are “owned” by the function, and those that are not. The only messages that are owned by the function are those that were created explicitly by the function, using the Pascal NEW function or the supplied functions NewQxxx and MsgCopy. The input messages to a function are NOT owned by that function. You should treat the input messages as being “read-only.”

The most important rule for handling messages properly is that a function should never attempt to modify, send, or dispose of messages which it does not own. You must also make sure that all of the messages that are created by the function are sent to an output queue (using SendMsg), stored on the private queue, or otherwise disposed of (as via DropMessage) before the function exits. (After this is done, the message is no longer owned by your function.) If the function exits without disposing of all of its owned messages, it will “eat” storage and may cause the PS 390 to crash as a result of exhausting available memory.

Some of the utility functions and procedures provided cause the values of the messages they take as arguments to become undefined (as `QSendCopyMsg`), or they set the pointer to the message to `NIL` (as `SendMsg`). You should be aware of these side-effects; refer to Section 8 for complete descriptions of the utility functions and procedures.

Messages are actually sent in the order that you make the calls to `SendMsg`, but not until the function has finished running. You should be careful to send messages in the correct order where necessary. For example, the outputs of the Bezier curve function in Section 6 are intended to be connected to a vector list. Output <1>, which clears the vector list, must be sent before output <2>, which appends to the vector list; otherwise the vector list would always be empty!

5.2.3 About Function States

Ordinarily, you need not be concerned about function states or the valid actions that a function can perform in each state, as long as the functions you write follow the template used in the examples in this manual. This information is provided for completeness.

A PS 390 function instance may be in one of several states at any given time. Transitions between the states are caused by the scheduler in the PS 390 system, or by calls to utility procedures when the function is running.

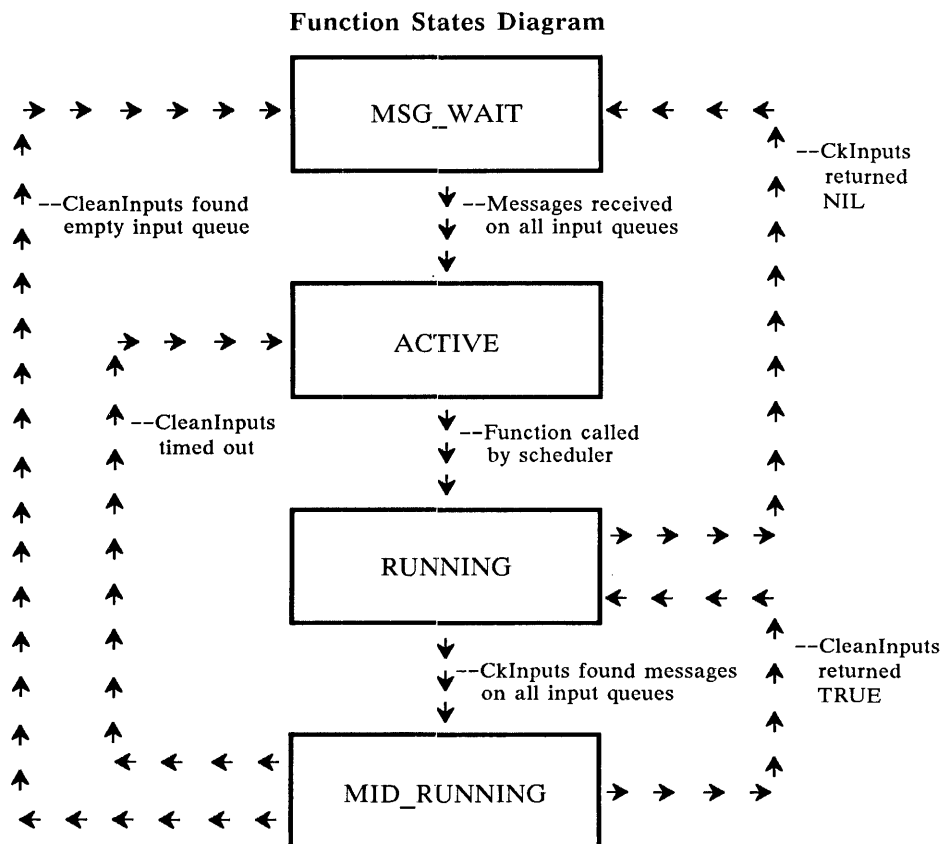
A function instance is in state `MSG_WAIT` while it is waiting for messages to arrive on all input queues. When all input queues have messages, the scheduler changes the state to `ACTIVE` and puts the instance on the list of functions that are ready to be activated. When the main procedure of the function is called by the scheduler, the state is changed to `RUNNING`.

The function instance must be in the state `RUNNING` when the utility procedure `CkInputs` is called. If messages are not available on all input queues, `CkInputs` returns `NIL` and the function state is changed back to `MSG_WAIT`. If there are messages on all inputs, the function state is set to `MID_RUNNING`.

While the function instance is in state `MID_RUNNING`, it should process its inputs and send outputs. When this is complete, the utility procedure `CleanInputs` must be called. This procedure can only be called from the `MID_RUNNING` state.

After disposing of the previous input messages, CleanInputs first checks to see if there are messages available on all input queues. If there are queues without messages, the state is changed to MSG_WAIT. Otherwise, a check is made to see if the function has been running longer than two milliseconds; if it has, then the state is changed to ACTIVE. (This gives other functions a chance to run.)

If the function has not been running longer than two milliseconds, its state is changed to RUNNING, allowing it to run again with the new set of input messages. The function may continue to execute as long as it is in the RUNNING or MID_RUNNING state. It must exit immediately if the state is changed to MSG_WAIT or ACTIVE. The following diagram illustrates the change of states in a function instance.

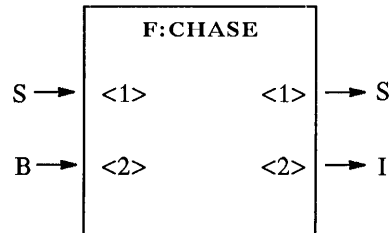


In the next section, *Writing Your Own Function*, you will be asked to write the Pascal procedure for a specific function. Before you begin this section, make sure you are familiar with the types of messages that can be handled and with the utility routines. This information is found in Section 8 of this manual.

5.3 Writing Your Own Function

The diagram below illustrates the function that you will be writing in this section.

Function



Description

F:CHCASE accepts ASCII character strings (qpacket data type) on input <1> and a Boolean value on input <2>. When input <2> is set to TRUE, the characters received on input <1> will be output as upper case. When input <2> is set to FALSE, the characters will be output as lower case. Output <1> takes the processed character string from input <1>. Output <2> sends out an integer that is the length of the string.

To write this function you must:

1. Name the program.
2. Include USERSTRUC.PAS.
3. Define the variables.
4. Check the inputs for data.
5. Check for the legitimate data types.
6. Allocate memory for the output messages.
7. Change case according to the value on input <2>.
8. Send the processed string to output <1>.
9. Send the count of the string to output <2>.
10. Flush the queues.
11. Check all queues for more data.

Because you will be using this program for exercises in compiling, linking, and downloading, for consistency it is suggested that you name your program:

```
SUBPROGRAM ChCase;
```

5.3.1 Exercise

Design and write the Pascal procedure for the function F:CHCASE, as previously described.

5.3.2 Feedback

The procedure for F:CHCASE is provided as an example. Please check your exercise against it to make sure you have included all the necessary steps. You can design your program in any number of ways, so long as it performs the necessary steps in the correct order.

```
SUBPROGRAM ChCase;

{$F=USERSTRUC.PAS}

PROCEDURE GenFunction;

  VAR
    inputs  : PtrUWFInQarray;
    length  : Integer;
    outmsg   : Ptrqdata;
    j,k      : Integer;

    {-----}
    { Utility functions for uppercasing and lowercasing a character }
    {-----}

  FUNCTION uppercase (ch : Char): Char;
  BEGIN
    IF (ch >= 'a') AND (ch <= 'z') THEN
      uppercase := chr (ORD(ch) - 32)
    ELSE
      uppercase := ch;
    END;

  FUNCTION lowercase (ch : Char): Char;
  BEGIN
    IF (ch >= 'A') AND (ch <= 'Z') THEN
      lowercase := chr (ORD(ch) + 32)
    ELSE lowercase := ch;
    END;
```

```

{-----}
{ Main body of UWF }
{-----}

BEGIN { GenFunction }
  inputs := CkInputs (1, 2);
  WHILE inputs <> NIL DO BEGIN
    IF inputs↑[1]↑.qtyp <> QPacket THEN
      Qillmessage (1)
    ELSE IF inputs↑[2]↑.qtyp <> QBoolean THEN
      Qillmessage (2)
    ELSE BEGIN

      {-----}
      { Allocate a new QPacket big enough to hold }
      { a the string. Then fill in the value and }
      { send the message from output <1>. }
      {-----}

      WITH inputs↑[1]↑ DO
        length := P_lth - P_beg + 1;
        outmsg := NewQPacket (QPacket, length);
        j := outmsg↑.P_beg;
        FOR k := inputs↑[1]↑.P_beg TO inputs↑[1]↑.P_lth DO BEGIN
          IF inputs↑[2]↑.b THEN
            outmsg↑.P_cnt[J] := uppercase (inputs↑[1]↑.P_cnt[k])
          ELSE
            outmsg↑.P_cnt[K] := lowercase (inputs↑[1]↑.P_cnt[k]);
          j := j + 1;
        END;
        SendMsg (outmsg, 1);

        {-----}
        { Send a message indicating the length of }
        { the string on output <2>. }
        {-----}

        outmsg := NewQInteger;
        outmsg↑.i := length;
        SendMsg (outmsg, 2);
        END;
      IF CleanInputs THEN
        inputs := CkInputs (1, 2)
      ELSE
        inputs := NIL;
      END;
    END. { GenFunction }

```

5.4 Compiling, Linking, and Naming the Function

The procedure you have just written must now be successfully compiled and linked. The processor in the PS 390 that executes functions is the Motorola M68000 microprocessor and your code must be compiled and linked by Motorola cross-software.

The next section provides simple instructions on using the files provided on magnetic tape for DEC VAX/VMS and DEC VAX/UNIX systems.

IBM VM/SP and IBM MVS/TSO users should briefly familiarize themselves with the information provided here, and then refer to section 9 in this manual for further instructions on using the cross-software. IBM MVS/TSO users should consult the Motorola manuals supplied with the cross-software.

Sections 9.1 thru 9.4 of this manual provide further instructions for each environment. If you are not operating in one of these environments, you will have to tailor the command files to your system or write your own.

Before continuing with this tutorial, make sure the cross-software and the files have been loaded on your system and that you have access to them.

Use the *Motorola Pascal User's Guide* as a reference to interpret any error messages produced by the compiler when your function is compiled.

5.4.1 Description of the Command Files for DEC VAX/VMS and UNIX

The command files provided for the DEC systems combine four specific tasks:

1. Compile your Pascal procedure with the Motorola cross-compiler.
2. Link your Pascal procedure with the MAIN program, USERLINK, to yield an S-record file. The S-record file is the definition of the userwritten function in a form that the PS 390 expects and will accept. USERLINK calls a procedure, GenFunction, which is the name of the body of the function that you have written. (Refer to the examples.)
3. Append a trailing semicolon, “;”, to the end of the S-record file. The semicolon must terminate any S-record file that is transferred to the PS 390.
4. Name your function. Before the S-record file can be downloaded to the PS 390 and the function instanced, it must be named. A name

“header” must be created that includes the number of inputs, the number of outputs, and the stack usage of the function. (The stack usage is the number of bytes that must be reserved on the stack for the function and includes the count of all utility routines that the function uses. Section 8 of this manual contains a listing of the stack usage for the user-written function utility routines.)

When the command file is called, it accepts as arguments the function name, number of inputs, number of outputs, and stack size.

There are several restrictions on the files that are provided as an aid in compiling, cross-linking and finally naming your function:

- The name of the file that will be compiled and linked using the command files must have the same name as the function.
- The provided command file will only accept one file name. This means that functions that use several files must be compiled, linked, and named under a modified version of the command file.

5.4.2 DEC VAX/VMS Command File

Before executing the following command, you should:

1. Set your default directory to the directory containing the source files for the function you want to compile and link. This directory should also contain copies of USERSTRUC.PAS and USERLINK.RO.
2. Edit your login.com file to contain an @XNAMES command.

As a convenience, a command file, XL.COM has been provided to compile and link your function and to produce the S-record file that is ready to be downloaded to the PS 390. All of the code for the function must be in a single .PAS file and the name given to the function is assumed to be the name of the file. To invoke this command, you should enter the command in the form:

```
$ XL <filename> <number inputs> <number outputs> <stack size>
```

In the above example, the brackets are provided to separate the arguments. When actually using the command, the brackets are not used and the arguments are separated by a single space. Error messages will be returned if any errors are encountered in the compiling, assembling, and linking process.

When this message is displayed:

```
<filename>.300 created
```

the cross-software has been successfully called, an S-record file has been produced, and the name header has been created. This file, <filename>.300, is ready to download to the PS 390.

If functions that you may write later contain code from more than one file, or if you want to include routines you have written in assembly language, refer to Section 9.1 for instructions. Refer to the *Motorola Pascal User's Guide* to interpret error messages that are generated at the time your code is compiled.

5.4.3 DEC VAX/UNIX Command Files

Before attempting to use the cross-software, you should edit your .cshrc file to “source” the file xnames, which defines the necessary aliases and shell variables. This allows the assembler, compiler, and linker to be used as described in the EXORMACS manuals.

Before executing the command described below, you should set your working directory to the directory containing the source files for the function you want to compile and link. This directory should also contain copies of userstruc.pas and userlink.ro. Since UNIX is case sensitive, remember to use consistent case for file names.

As a convenience, a shell script xl has been provided to compile and link your user-written function and to produce the S-record file ready to be downloaded to the PS 390. All of the code for the function must be contained in a single .pas file, and the name of the function is assumed to be the name of the file. To invoke this shell script, you should enter the command in the form:

```
% xl <filename> <number inputs> <number outputs> <stack size>
```

In the above example, the brackets are provided to separate the arguments. When actually using the command, the brackets are not used and the arguments are separated by a single space. Error messages will be returned if any errors are encountered in the compiling, assembling, and linking process. Refer to the *Motorola Pascal User's Guide* to interpret these error messages.

When this message is displayed:

```
<filename>.300 created
```

the cross-software has been successfully called, an S-record file has been produced, and the name header has been created. This file, <filename>.300, is ready to download to the PS 390. If functions that you may write later contain code from more than one file, or if you want to include routines you have written in assembly language, refer to Section 9.2 for instructions.

5.4.4 Instructions for IBM Systems

If you are using an IBM system, refer to the following appendices (or manuals) for instructions.

IBM VM/SP	Section 9.3 for the names of the files appropriate for your system, information on the use of the cross-software on your host system, and example files that execute the cross-software.
-----------	--

Section 9.7 for instructions on how to build the name header that will be downloaded to the PS 390 prior to the S-record file.

IBM MVS/TSO	Section 9.4 for the names of the files appropriate for your system. Please refer to the Motorola cross-software manuals for instructions on using the cross-software on your system.
-------------	--

Section 9.7 for instructions on how to build the name header that will be downloaded to the PS 390 prior to the S-record file.

5.4.5 Exercise

Compile, link, and name the function F:ChCase. If you are using the command files, remember that the name of the function must be the same as the name of the file that contains the Pascal procedure; i.e., your file name should be ChCase.pas. To successfully complete this exercise, complete the steps listed for your operating environment.

For DEC VAX/VMS or UNIX:

1. Invoke the command file appropriate for your host system.
2. Enter the parameters for the function, F:CHCASE, including name, number of inputs, number of outputs, and stack size. (1000 is a reasonable estimate for the stack size of any function similar in size to F:CHCASE.)

For IBM VM/SP:

1. Compile and link your function using the instructions provided in Section 9.3.
2. Build the function header line (refer to Section 9.6).
3. Append your file with the trailing semicolon “;”.

For IBM MVS/TSO:

1. Compile and link your function using the instructions provided in the Motorola manuals and Section 9.4.
2. Build the function header line (refer to Section 9.6).
3. Append your file with the trailing semicolon “;”.

5.4.6 Feedback

The following example is provided to illustrate what should have been entered at your host terminal to call the Motorola cross-software successfully and to create the name header for your function for DEC systems:

VAX/VMS:

```
$ XL CHCASE 2 2 1000
```

VAX/UNIX:

```
% x1 ChCase 2 2 1000
```

IBM systems users should use the example files provided in section 9 to check their exercise.

5.5. Transferring the Function to the PS 390

After your Pascal procedure has been compiled and linked with the main program USERLINK, the S-record file output by the linker must be modified to include a function header line and to terminate with a semicolon. If

you have compiled and linked your function using the command files previously described, this will be done for you. See Section 9.6 for a description of the header line format.

If you are using VAX/VMS, UNIX, or any other ASCII system over an RS-232 asynchronous line, the S-record file can be downloaded by including the routing bytes to access the appropriate channel. Input on this channel is sent to a PS 390 function that writes the new function into mass memory.

For IBM systems or high-speed lines, these channels are accessed by using the Utility Routines provided by the PS 390 Graphics Support Routines (GSRs), rather than using the ASCII routing bytes. Both transfer methods will be discussed in the following section.

5.5.1 Using Routing Bytes to Transfer the S-Record File

If you are not familiar with the use of routing bytes in the PS 390, please refer RM7 of the *PS 390 Document Set*. In general, routing bytes are used to toggle between different communication channels in the PS 390 system. In downloading the S-record files for user-written functions, the channel which loads the functions to mass memory must be accessed. The routing bytes that open this channel are $\uparrow\backslash 6$, where $\uparrow\backslash$ is the field separator character (decimal 28) and 6 designates the channel for loading user-written functions into memory.

Once the user-written functions have been transferred to the PS 390, you should change the channel back to the terminal emulator so that any error messages can be intercepted and displayed on the PS 390 screen. The routing bytes for the terminal emulator are $\uparrow\backslash >$.

The routing bytes and S-record file can be sent to the PS 390 in number of ways. The suggestions that follow outline some of the normal communication methods available between the PS 390 and an ASCII host system.

1. A host-system command file can be built that uses standard host transfer commands to send the mass memory routing bytes ($\uparrow\backslash 6$), then the file containing the name header and S-record file, and finally the routing bytes that open the communication channel to the terminal emulator.
2. Individual files containing the routing bytes can be built and then copied to the PS 390 by a command file. The file transfers the opening sequence of routing bytes, the S-record file, and finally the routing bytes to change the channel.

3. The file containing the S-record file can be edited using host facilities and the routing bytes can be included at the top and at the end of the file. The file can then be sent to the PS 390. The routing bytes are stripped out once the file is passed to communication functions in the PS 390, so they would not be the final code that is used by the function when it is instanced.

5.5.2 Using the Graphics Support Routines to Transfer the S-Record File

For any non-ASCII system, it is recommended that the utility routines in the Graphics Support Routines (GSRs) be used to access data channels and transfer the S-record file. The GSRs are provided in FORTRAN, Pascal and UNIX/C. If you are not familiar with the GSRs, refer to RM4 of the *PS 390 Document Set*.

The channel parameter 7 should be used with the utility routine, PMUXG, to access the channel to mass memory. It is recommended that the channel to the terminal emulator, 15, be reconnected after the transfer is complete so that error messages will be displayed on the PS 390 screen.

The following Pascal program illustrates how the GSRs can be used to transfer the S-record file from the host system to the PS 390. This example illustrates the use of the GSRs in an IBM VM/SP environment.

```
File:  SRECSND PASCAL  *

Program SRecSnd (input,output,srecfile);

CONST
    %INCLUDE PROCONST

TYPE
    %INCLUDE PROTYPES

VAR
    srecfile   :   Text;
    istr       :   String (256);
    crlfa      :   Packed array (.1..2.) of char;
    crlf       :   String(2);
```

```

%INCLUDE PROEXTRN
PROCEDURE err (errnum : integer);
BEGIN
  writeln( 'got error: ', errnum );
END;

BEGIN
  pattach('junk',err);
  reset( srecfile);
  crlfa (.1.) : CHAR (13);
  crlf := STR(crlfa);
  pmuxg(7,err);
  WHILE NOT EOF (srecfile) DO
    BEGIN
      readln (srecfile, istr);
      pputgx (istr,err);
      pputgx (crlf, err);
    END;
    writeln;
    pmuxg(15,err);
    pdetach(err);
  END.

```

5.5.3 Exercise

Using any of the previously described methods, transfer the file ChCase.300 (renamed after it was compiled and linked) from your host system to the PS 390.

5.5.4 Feedback

The only way to check and see if your file transferred successfully is to try to instance the function. This is done by entering Command mode on the PS 390 (refer to IS3 of the *PS 390 Document Set* for entering the communication modes) and instancing the function using the standard PS 390 command:

```
instance name := F:CHCASE;
```

If no error message is returned, the function was successfully downloaded and now resides in mass memory in the PS 390.

If the downloading process fails, the PS 390 may crash and require rebooting. After rebooting, attempt to download the file at least once more.

If the PS 390 still crashes, check the following:

1. The correct file name was used in the transferring process.
2. The correct routing bytes or channel parameters were used.

If you are compiling and linking using the command files provided, the information and format of the S-record file are valid. If you are not using the command files, check for the following:

1. Correct syntax in the name header, including adequate stack size.
2. Trailing semicolon at the end of the file.
3. Correct routing bytes or channel parameters.

5.6 Instancing the Function

When the function has been successfully transferred to the PS 390, it is instanced using the standard PS 390 command:

```
instance name := F:user-written function name;
```

Once the function is resident in mass memory, there are several restrictions that apply to all user-written functions:

1. Initializing the system with the global INIT command, or using the INIT NAMES command will destroy not only all instances of the function, but also the body of the function. The function would have to be again transferred down from the host before it would be available for use on the PS 390. Functions can be protected from the INIT commands. The procedure for doing this is described in Section 7.
2. Naming anything with the same name given to a user-written function will cause the function to be replaced by the new entity of that name. In particular, note that the command

```
ChCase := F:CHCASE;
```

will destroy the code for the function F:CHCASE.

With the exception of those restrictions, the user-written function will respond as any intrinsic function resident in the PS 390.

5.7 Debugging User-Written Functions

The debugging environment on the PS 390 is less powerful than that used for debugging programs on the host computer. There is no symbolic debugger, and no way to include “writeln” statements inside the function to examine intermediate results and trace its execution. The function you are debugging is very much like a black box: you can see what goes in and what comes out, but you cannot look inside it.

The standard technique for debugging a user-written function is to instance it and connect all outputs to F:PRINT functions, and from there to the terminal emulator or LABELS or CHARACTERS structures. Then, you SEND messages to the input queues of the function and examine the results. If the user-written function does not behave as expected, it is possible to replace the code for the function (by recompiling and relinking on the host and downloading the new S-record file) without losing the function instance or connections to and from it.

If the function does not produce any outputs at all, make sure that it is receiving messages on all of its input queues. Remember that all queues default to being active queues unless you use SETUP CNESS to make them constant queues.

If the function is only sending some of the output messages you expect, look for bugs in the body of the function. Make sure that SendMsg is being used to send the messages to the appropriate output queues.

When the values of the output messages are incorrect, it is sometimes useful to modify the function temporarily to have additional outputs for sending intermediate results. By examining these values, it is possible to isolate the source of the problem. Once the problem has been fixed, the extra outputs can be removed.

There are two common problems that cause running a user-written function to crash the PS 390. If the PS 390 crashes immediately (as soon as the function is activated), it is probably because the stack size you specified when you created the S-record file is not big enough. Try increasing this value. (Use the Stack Usage list in Section 8 of this manual to help determine stack size. The stack requirements are given for the utility routines.)

Another cause of an immediate crash is when the number of inputs specified in the name header line is not correct.

If the crash occurs randomly after the function has been run, it is probably the result of trying to send or otherwise dispose of a message which is not owned by the user-written function. The crash occurs when the true owner of the message tries to access it. In this case, you should examine the code for the function for proper use of messages. Section 8 of this manual contains a list of common crash messages and their probable cause.

Remember that while your function is running, nothing else will. If the PS 390 seems to “hang” when the function is activated (i.e., it does not respond to the keyboard or other devices), look for an infinite loop in the function. Problems with message ownership can sometimes cause an infinite loop if you are trying to use a message whose value has become undefined (as by using SendMsg or QSendCopyMsg) as the upper limit of a FOR loop.

Another common problem is when the function “eats” memory. This is usually most noticeable after the function has been run many times. In this situation, examine the function closely to be sure that all of the messages it creates are being sent as outputs, or otherwise disposed of, before the function exits.

5.7.1 Exercise

Create an instance of F:CHCASE. Connect it to a network that will allow you to examine the contents of the messages that are sent out of the function to determine if it is working correctly. To do this you will have to:

1. Create an instance of F:CHCASE.
2. Create an instance of F:PRINT for each output of F:CHCASE.
3. Create a label node that will accept and display the character string from F:PRINT.
4. Create a network that will feed the messages from the output of F:CHCASE through the PRINT function to the label node.
5. SEND messages to the input queues of F:CHCASE and examine the results.

5.7.2 Feedback

The strings displayed on the PS 390 screen should accurately reflect the case of the characters in the string, as determined by the Boolean value on input <2> of F:CHCASE.

5.8 Conclusion

This completes Section 5, the User-Written Functions Tutorial. By this time, you should be able to construct, compile and link, download, and instance a simple function. Section 6, Examples of More Advanced Ideas, moves from a strictly tutorial format to a format that demonstrates by examples some of the more advanced programming capabilities that can be used when writing your own functions.

Section 7 provides instructions for transferring user-written functions to the PS 390 Firmware diskettes, allowing them to load with the system, and PS 390 commands that protect user-written functions from global INITIALIZE commands. It also contains information on the use of the PS 390 Debugger.

Section AP6

More Advanced Ideas

This section illustrates, through examples and text, how to write more complex functions. There are four major examples, each illustrating a different type of function. The functions illustrate the following concepts:

F:MAG

How to handle more than one Qdata type on the same input queue.

F:COUNT

How to use the Set_Cness utility routine and private data queues.

F:BEZIER

How to write a function with a variable number of input queues.

F:SPIRO

How to make use of the user-defined Qdata type.

Before proceeding with this section, you should be familiar with:

- The concepts presented in section AP5.
- The information provided in the section AP8 of this manual.

Should you need further exercise in writing functions, it is recommended that you use this section in the following manner:

1. Examine the initial introduction to each example and the description of the function.
2. Using the section AP8 of this manual, write a procedure to support the described function.
3. Check your code against the examples provided.
4. Compile, download, and instance the function.
5. Try it out.

6.1 Example I - Handling Different Message Types on the Same Queue

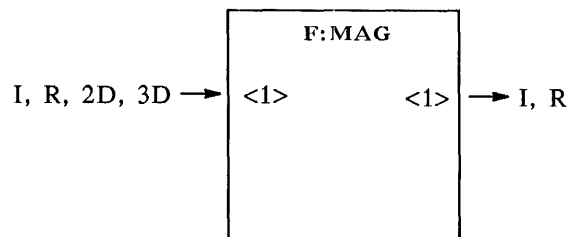
F:MAG

The magnitude function, F:MAG, is an example of a function that can handle several types of messages on the same input queue. This function will calculate the absolute value, if the input message is of type integer or real, or the length, if the input is a 2D or 3D vector.

If a function has two or more inputs that can take different message types, it is often necessary to make additional checks to make sure that the received messages are compatible types. If this is not the case, there is a utility procedure, QIncompatMsgs, provided to signal the error.

F:MAG illustrates how the body of a function that can accept different types of inputs usually takes the form of a single IF ELSE IF ELSE statement. Each IF clause tests for a valid combination of inputs, with the final ELSE clause being used to flag an error.

Function



Description

This function calculates the absolute value (if integer or real) or magnitude (if a vector) of the input received.

Example

```
SUBPROGRAM uwfmag;  
  
{$F=USERSTRUC.PAS}  
  
PROCEDURE GenFunction;  
  
VAR  
    inputs : PtrUWFInQarray;
```

```

outmsg : Ptrqdata;
temp   : double;
BEGIN {GenFunction}
inputs := CkInputs (1, 1);
WHILE inputs <> NIL DO BEGIN
  IF inputs↑[1]↑.qtyp = QInteger THEN BEGIN {send absolute value}
    outmsg := NewQInteger;
    outmsg↑.i := abs(inputs↑[1]↑.i);
    SendMsg (outmsg, 1);
  END
  ELSE IF inputs↑[1]↑.qtyp = QReal THEN BEGIN {send absolute value}
    outmsg := NewQReal;
    outmsg↑.r := inputs↑[1]↑.r;
    FpAbs (outmsg↑.r);
    SendMsg (outmsg, 1);
  END
  ELSE IF inputs↑[1]↑.qtyp = QVec2 THEN BEGIN {send sqrt (x*x + y+y)}
    outmsg := NewQReal;
    FCMultiply (inputs↑[1]↑.v4[0], inputs↑[1]↑.v4[0], outmsg↑.r);
    FCMultiply (inputs↑[1]↑.v4[1], inputs↑[1]↑.v4[1], temp);
    FCAdd (outmsg↑.r, temp, temp);
    FCSqroot (temp, outmsg↑.r);
    SendMsg (outmsg, 1);
  END
  ELSE IF inputs↑[1]↑.qtyp = QVec3 THEN BEGIN {send sqrt (x*x + y* y}
                                         {z*z)          }
    outmsg := NewQReal;
    FCMultiply (inputs↑[1]↑.v4[0], inputs↑[1]↑.v4[0], outmsg↑.r);
    FCMultiply (inputs↑[1]↑.v4[1], inputs↑[1]↑.v4[1], temp);
    FCAdd (outmsg↑.r, temp, outmsg↑.r);
    FCMultiply (inputs↑[1]↑.v4[2], inputs↑[1]↑.v4[2], temp);
    FCAdd (outmsg↑.r, temp, temp);
    FCSqroot (temp, outmsg↑.r);
    SendMsg (outmsg, 1);
  END
  ELSE                                     {anything else is illegal}
    QillMessage (1);
  IF Cleaninputs THEN
    inputs := Ckinputs (1, 1)
  ELSE inputs := NIL;
END;
END. {GenFunction}

```

6.2 Example II - SET_CNESS and Private Queues

F:COUNT The function F:COUNT is a simple counter function. Input <1> is the trigger queue. A Boolean value of TRUE causes the counter to be reset to the value received on input <2>. A value of FALSE causes the current value of the counter to be incremented. Input <2> is a constant queue.

The utility procedure Set_Cness is used here to “hard-code” the cness of the queues. Usually, it is preferable to rely on using the SETUP CNESS command to establish the cness of the queues for each individual instance of a function. If you use the Set_Cness utility procedure, you cannot also use SETUP CNESS on the same queue. In F:COUNT, however, it is hard to imagine a situation where you would not want to have the initial value of the counter a constant queue, so you would want to use the Set_Cness utility.

If a function uses the Set_Cness procedure, the call should appear at the very beginning of the function body. Trying to change the Cness of a queue back and forth in the middle of the function will probably not do anything useful.

F:COUNT also illustrates the use of the private queue to store the current value of the counter. Ordinarily, a user-written function has no global variables or other permanent information that remain from one activation of the function to the next. Since being able to “save state” is sometimes required for a function to perform its proper task, each user-written function is provided with a private queue to contain permanent information.

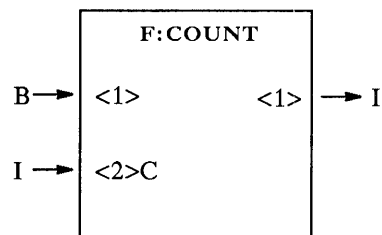
The name “private queue” is used because the only way messages can be placed there is from inside the function itself; you cannot SEND to the private queue of a function.

When a function is instanced, the private queue is initially empty. Therefore, one of the very first things a function that uses the private queue should do is check to see whether or not there is already a message on the queue, using the utility function CkPrivate. This function will return a pointer to the message if it exists. If the queue is empty, create a new message of the appropriate type and use the function SavePrivate to store it in the private queue. Once a message has been saved on the private queue, it remains there permanently. The message is owned by the function and may be modified as necessary.

Note that the private queue really is a queue, and may contain more than one message. You can chain several messages together into a linked list by storing a pointer in the NEXT field in the Qdata record. But be careful--messages on the private queue are the only instance where accessing the NEXT field directly will not cause huge amounts of trouble!

In the case of F:COUNT, the private queue contains a single message of type QInteger.

Function



Description

This function is a simple counter. The private queue is used to maintain the last value of the counter.

Input <1> is the trigger queue. When a message of TRUE is received, the counter is reset to the value on input <2>. When a message of FALSE is received, the counter is incremented. The current value of the counter is sent on output <1>.

Example

```
SUBPROGRAM uwfcount;

{$F=USERSTRUC.PAS}

PROCEDURE GenFunction;

VAR
  inputs  : PtrUWFInQarray;
  outmsg  : Ptrqdata;
  status  : Ptrqdata;
```

```

BEGIN  {GenFunction}

    {-----}
    { Set input <2> to be a constant queue.  This }
    { is done because it does not make sense to  }
    { have this an active queue.                }
    {-----}

Set_Cness (2, TRUE);

    {-----}
    { Get the inputs to the function and process. }
    {-----}

status := NIL;
inputs := CkInputs (1, 2);
WHILE inputs <> NIL DO BEGIN

    {-----}
    { First the usual check for valid inputs.    }
    {-----}

    IF inputs↑[1]↑.qtyp <> QBoolean THEN
        Qillmessage (1)
    ELSE IF inputs↑[2]↑.qtyp <> QInteger THEN
        Qillmessage (2)
    ELSE BEGIN

        {-----}
        { Now that's taken care of, you need to get }
        { the message from the private queue before }
        { you can continue.  If the private queue   }
        { is empty, allocate and initialize a new   }
        { message.                                  }
        {-----}

        IF status = NIL THEN BEGIN
            status := CkPrivate;
            IF status = NIL THEN BEGIN
                status := NewQInteger;
                SavePrivate (status);
                status↑.i :=inputs↑[2]↑.i;
            END;
        END;
    END;
END;

```



```

{-----}
{ Then you can use the message from the      }
{ private queue to determine the value to    }
{ be output.                                }
{-----}

```

```

IF inputs↑[1]↑.b THEN
    status↑.i := inputs↑[2]↑.i
ELSE
    status↑.i := status↑.i + 1;
    outmsg := NewQInteger;
    outmsg↑.i := status↑.i; SendMsg (outmsg, 1);
END;

IF Cleaninputs THEN
    inputs := ckinputs (1, 2)
ELSE
    inputs := NIL;
END;
END. { GenFunction }

```

6.3 Example III - Variable Number of Input Queues

F:BEZIER(N)

The Bezier curve function, F:BEZIER(N), is an example of a user-written function which has a variable number of inputs. Input <1> is an integer indicating how many points on the curve are to be calculated. Inputs <2> through <N> are points (3D vectors) which define the Bezier curve. Output <1> of the function is intended to be connected to the <clear> input of a vector list, and output <2> should be connected to the <append> input of the same vector list.

The body of the Bezier function is very similar to the code for any ordinary user-written function. The only difference is that, since you do not know how many inputs the instance of the function has ahead of time, you must call the utility procedure My In Out to find out. The call to this procedure should be placed at the very beginning of the function, before any calls to Set Cness, and before the initial call to CkInputs.

To indicate that a function has a variable number of inputs and/or outputs, you should specify the corresponding parameter as 255 in the header line of the S-record file. For example, since the Bezier function has a variable number of inputs and only 2 outputs, the header line should indicate 255 inputs and 2 outputs. On VAX/VMS, the command to compile and link the Bezier function is:

```
$ XL Bezier 255 2 5000
```

Then, when you instance the function, you must specify the actual value for N, as

```
<instance_name> := F:<function_name>(N);
```

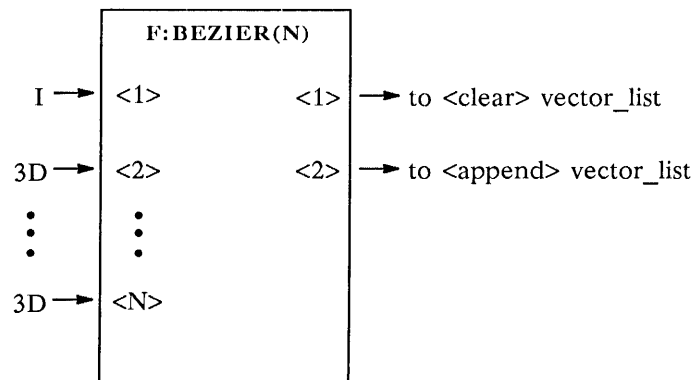
In this example, the following command will create an instance of the F:BEZIER function with 10 inputs:

```
drawit := F:Bezier(10);
```

If the function has a variable number of inputs, N specifies the number of inputs. If the function has a variable number of outputs, N specifies the number of outputs. If both the number of inputs and the number of outputs are variable, the function will have N inputs and N outputs.

Another interesting point about the Bezier function is that it is very important to send the output messages in the correct order. Since the messages are actually delivered in the same order that they were sent within the function code, you must be careful that the clear message is sent on output <1> before any vectors are sent on output <2>. Otherwise, the vector list which receives these messages would always appear empty.

Function



Description

F:BEZIER evaluates a series of points on a Bezier curve. Input <1> is an integer indicating how many points on the curve are to be calculated. Inputs <2> through <N> are points (3D vectors) which define the vertices of the Bezier curve. Output <1> of the function is intended to be connected to the <clear> input of a vector list, and output <2> should be connected to the <append> input of the same vector list.

Example

```
SUBPROGRAM uwf bezier;

{$F=USERSTRUC.PAS}

PROCEDURE Genfunction;

TYPE
    varray = ARRAY[1..MaxInputQueues] OF vector;

VAR
    ins, outs : Int16;
    inputs    : PtrUWFInQarray;
    i, j      : Integer;
    npnts     : Integer;
    outmsg    : Ptrqdata;
    error     : Boolean;
    vertices  : Varray;
    t, delta  : Double;

{-----}
{ A function to evaluate the coordinates of a point on }
{ a Bezier curve defined by "vertices" at parameter   }
{ value "t"                                           }
{-----}

FUNCTION eval_Bezier (VAR vertices : Varray;
                     nvert: Integer;
                     t: Double): vector;

VAR
    j, k, m : Integer;
    temp     : Double;
BEGIN
    FOR j := nvert-1 DOWNTO 1 DO { loop over iterations }
        FOR k := 1 TO j DO { loop over each vertex }
```

```

        FOR m := 0 TO 2 DO BEGIN      { loop over x,y,z      }
            FCSubtract (vertices[k+1,m], vertices[k,m], temp);
            FCMultiply (temp, t, temp);
            FCAdd (vertices[k,m], temp, vertices[k,m]);
        END;
    eval_Bezier := vertices[1];
END;

{-----}
{ Main body of UWF      }
{-----}

BEGIN { GenFunction }

    My_in_out (ins, outs);
    inputs := CkInputs (1, ins);
    WHILE inputs <> NIL DO BEGIN
        error := FALSE;
        IF inputs↑[1]↑.qtyp <> QInteger THEN BEGIN
            error := TRUE;
            Qillmessage (1);
        END
        ELSE IF inputs"[1]".i < 1 THEN BEGIN
            error := TRUE;
            Qillvalue (1);
        END;
        FOR i := 2 TO ins DO
            IF inputs↑[i]↑.qtyp <> QVec3 THEN BEGIN
                error := TRUE;
                Qillmessage (i);
            END;
        IF (inputs <> NIL) AND (NOT error) THEN BEGIN

            {-----}
            { Send the CLEAR message out first.  You need }
            { to save the value of input 1 before doing   }
            { QSendCopyMsg, because once this is done the }
            { function no longer owns that message.       }
            {-----}

            npnts := inputs↑[1]↑.i - 1;
            QSendCopyMsg (1, 1);

            {-----}
            { Now calculate points on the curve and send }
            { them on output 2.                          }
            {-----}

```

```

FCInt2Double (npnts, delta);
FOR j := 0 TO npnts DO BEGIN
  FOR i := 2 TO ins DO
    vertices[i-1] := inputs↑[i]↑.v4;
  FCInt2Double (j, t);
  FCDivide (t, delta, t);
  outmsg := NewQVector (QVec3);
  outmsg↑.v4 := eval_Bezier (vertices, ins-1, t);
  SendMsg (outmsg, 2);
  END;
END;
IF Cleaninputs THEN
  inputs := ckinputs (1, ins)
ELSE
  inputs := NIL;
END;
END. { GenFunction }

```

6.4 Example IV - User-Defined Qdata Type

F:SPIRO

F:SPIRO is a function which behaves like a spirograph toy. A spirograph consists of two gears, an inner wheel and an outer ring. A pen fixed to the inner gear traces a pattern as it is rotated inside the outer ring.

Input <1> of the function will accept any message; it serves to trigger the function. The remaining inputs are constant queues. Input <2> is an integer specifying the number of teeth on the inner wheel, and input <3> specifies the number of teeth on the outer ring. Input <4> is a real number indicating the offset of the pen from the edge of the inner wheel.

Output <1> is intended to be connected to the <clear> input of a vector list, and output <2> to the <append> input of the same vector list. A Boolean TRUE is sent on output <3> to indicate that there are additional points on the curve to be calculated. Output <4> is a boolean TRUE that is sent only when all points on the curve have been calculated.

Instead of calculating all of the points which define the curve at once (as the F:BEZIER function does), F:SPIRO will only output one point each time it is activated. The state information is saved on the private queue so that the next time the function is activated, it can pick up where it left off. Output <3> can be connected back to input <1> to “reschedule” the function.

This approach is useful because it allows the computations to be “interruptible.” If the spirograph function were allowed to run continuously until the entire curve was calculated, it could take up to several minutes to complete (depending on the complexity of the curve). During this time, nothing else would be able to run on the PS 390. Breaking up the computation allows other PS 390 functions to run normally--including updating of the vector list to which the spirograph function is connected.

A special message type was defined to save the state information on the private queue for the spirograph function. One of the Qdata types, QuserType, is reserved for this purpose. A copy of USERSTRUC.PAS was modified to include a definition of the record type StateType, which contains fields to store the information that must be saved from one activation of the function to the next. The declaration of the Qdata record type was then modified so that QuserType messages contain a field (StateInfo) of this type. The spirograph function can then use the Pascal procedure NEW to create new QuserType messages.

The following is an excerpt from SPSTRUC.PAS (modified USERSTRUC.PAS) illustrating the definition of QuserType messages.

```

TYPE
  .
  .
  .
  statetype = RECORD
    newdata          : Boolean;
    maxi, maxo       : Integer;
    currenti, currento : Integer;
    offset           : Double;
    radius1, radius2 : Double;
    dthetai, dthetao : Integer
  END;
  .
  .
  .
  Qdata =
    RECORD
      Next: Ptrqdata;      { next message in a list of messages }
      CASE Qtyp: Qdtype OF { type of message }
        .
        .
        .
      QuserType:
        ( StateInfo : StateType );
      END ; { Qdata }

```

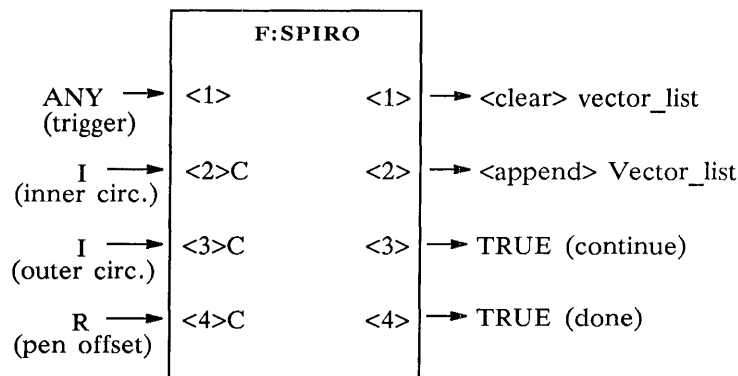
You can also use QuserType messages for communication between a set of user-written functions. These messages can be sent as output and received as input, just like any other message types. QuserType messages will also be handled correctly by any other function which accepts “any” message on the appropriate input, such as F:SYNC.

Your function is responsible for correctly initializing QuserType messages. If the message type is to be shared by several functions, it might be convenient to add a procedure to the modified copy of USERSTRUC.PAS which creates and initializes new QuserType messages, similar to the predefined NewQxxx utility procedures.

Qusertype Qtyp fields must be explicitly filled in by your program. Also, if the QuserType you define has fields in it that are pointers to other blocks, your function is responsible for disposing of these blocks. They must be disposed of before disposing of the QuserType message. The DropMessage utility routine (used to dispose of messages) should be called after you dispose of any such blocks.

If your QuserType message is sent to a PS 390 intrinsic function that accepts “any” message on an input, any block pointed to in the internal fields of the QuserType will not be properly disposed of. If you must include pointers in your QuserType definition, make sure they are properly handled.

Function



Description

This UWF is a spirograph function. A spirograph consists of two circular gears, an inner wheel rotating inside a fixed outer ring. A pen is fixed to

the inner gear at some offset from its circumference, so that it draws a pretty picture.

Sending a message to input <1> triggers the function. Input <2> is the number of teeth on the inner gear, and input <3> is the number of teeth on the outer wheel. Input <4> is the distance the pen is offset from the circumference of the inner gear.

The function outputs one line segment at a time. A value of TRUE is sent from output <3> to indicate that the function should be rescheduled. This output may be connected back to input <1>. TRUE is sent from output <4> when the curve is complete.

The curve is constructed as follows. *Maxi* and *maxo* refer to the number of teeth on the inner and outer gears, respectively, and *currenti* and *currento* refer to the pair of teeth that are currently meshing. *Dthetai* and *dthetao* are the angles subtended by a single gear tooth on the inner and outer gears, respectively. *Radius1* is the distance from the center of the fixed ring to the center of the inner wheel, and *radius2* is the distance from the center of the inner wheel to the pen. First, the angles of the two teeth that are currently meshing are found. The angles *thetai* and *thetao* are both relative to a fixed coordinate system. Then, the (x,y) coordinates of the pen location are given by:

$$\begin{aligned}x &= \text{radius1} * \cos(\text{thetao}) + \text{radius2} * \cos(\text{thetai}) \\y &= \text{radius1} * \sin(\text{thetao}) + \text{radius2} * \sin(\text{thetai})\end{aligned}$$

Note that the SinCos utility procedure expects the angle to be an integer from 0 to 65536 ($2 * \pi$), so you use this format for all the angles throughout the function.

Example

```
SUBPROGRAM uwfspiro;

{$F=SPSTRUC.PAS }           {USERSTRUC.PAS with Quserdata type defined}

PROCEDURE GenFunction ;

VAR
inputs   : PtrUWFWUFInQarray;
outmsg   : PtrQdata;
```



```

state   : PtrQdata;
si, ci  : Double;
so, co  : Double;
thetai  : Integer;
thetao  : Integer;

```

```

{-----}
{ A utility procedure to fetch information      }
{ stored on the private queue.  If nothing is  }
{ on the private queue, or if the information  }
{ is obsolete, reinitialize the state         }
{ information using the input msgs.           }
{-----}

```

```

PROCEDURE fetch_state_information;

```

```

  CONST

```

```

    pi2exp = 1027;          { Exponent part of 2 * pi }
    pi2man = 1686628288;    { Mantissa part of 2 * pi }

```

```

  VAR

```

```

    pi2, temp : Double;

```

```

  BEGIN

```

```

                                {-----}
                                { If the private queue is empty, create      }
                                { and store a new message.                     }
                                {-----}

```

```

    state := CkPrivate;

```

```

    IF state = NIL THEN BEGIN

```

```

        NEW (state, QuserType);
        state↑.Qtyp := QuserType;
        state↑.stateinfo.newdata := TRUE;
        SavePrivate (state);
    END;

```

```

                                {-----}
                                { If you are beginning a new curve, store      }
                                { the new set of constants.  Calculate          }
                                { the radii of the two gears and find the      }
                                { angle subtended by a single tooth, as        }
                                { well as resetting other state variables.      }
                                {-----}

```

```

    IF state↑.stateinfo.newdata THEN

```

```

        state↑.stateinfo.maxi := inputs↑[2]↑.i; { copy input constants }
        state↑.stateinfo.maxo := inputs↑[3]↑.i;

```

```

state↑.stateinfo.offset := inputs↑[4]↑.r;
pi2.m := pi2man;          { find gear radii }
pi2.c := pi2exp;
FCInt2Double (state↑.stateinfo.maxi, temp);
FCDivide (temp, pi2, state↑.stateinfo.radius2);
FCInt2Double (state↑.stateinfo.maxo, temp);
FCDivide (temp, pi2, state↑.stateinfo.radius1);
FCSubtract (state↑.stateinfo.radius1, state↑.stateinfo.radius2,
            state↑.stateinfo.radius1);
FCSubtract (state↑.stateinfo.radius2, state↑.stateinfo.offset,
            state↑.stateinfo.radius2);
state↑.stateinfo.dthetai := 65536 DIV maxi; { angles of one }
                                           { gear tooth }

state↑.stateinfo.dthetao := 65536 DIV maxo;
state↑.stateinfo.currento := 0; { set current tooth counter }
END;

END;

{-----}
{ Main body of UWF. }
{-----}

BEGIN { GenFunction }
      {-----}
      { Establish constant queues. }
      {-----}

Set_Cness (2, TRUE);
Set_Cness (3, TRUE);
Set_Cness (4, TRUE);
      {-----}
      { Check for valid inputs. }
      {-----}

inputs := CkInputs (1, 4);
WHILE inputs <> NIL DO BEGIN

  IF inputs↑[2]↑.qtyp <> QInteger THEN
    Qillmessage (2)
  ELSE IF inputs↑[3]↑.qtyp <> QInteger THEN
    Qillmessage (3)
  ELSE IF inputs↑[4]↑.qtyp <> QReal THEN
    Qillmessage (4)
  ELSE IF inputs↑[2]↑.i <= 0 THEN
    Qillvalue (2)
  ELSE IF inputs↑[3]↑.i <= 0 THEN
    Qillvalue (3)

```

```

ELSE IF inputs↑[2]↑.i >= inputs↑[3]↑.i THEN
    Qillvalue (2)
ELSE BEGIN
    {-----}
    {Get state info from the private queue.  }
    {-----}

    Fetch_State_Information;

    {-----}
    { If you are starting a new figure, do   }
    { something special to initialize it.    }
    {-----}

    IF state↑.stateinfo.newdata THEN BEGIN
        outmsg := NewQInteger;
        outmsg↑.i := state↑.stateinfo.maxo;
        SendMsg (outmsg, 1);
        outmsg := NewQVector (Qvec2);
        FCAdd (state↑.stateinfo.radius1, state↑.stateinfo.radius2,
            outmsg↑.v4[0]);
        FCInt2Double (0, outmsg↑.v4[1]);
        SendMsg (outmsg, 2); state↑.stateinfo.newdata := FALSE;
        END;
        {-----}
        { Calculate the next point on the figure. }
        {-----}

        state↑.stateinfo.currenti := (state↑.stateinfo.currenti + 1)
            MOD state↑.stateinfo.maxi;
        state↑.stateinfo.currento := (state↑.stateinfo.currento + 1)
            MOD state↑.stateinfo.maxo;
        thetao := state↑.stateinfo.currento *
            state↑.stateinfo.dthetao;
        thetai := thetao - (state↑.stateinfo.currenti *
            state↑.stateinfo.dthetai);
        Sincos (thetao, so, co);
        FCMultiply (so, state↑.stateinfo.radius1, so);
        FCMultiply (co, state↑.stateinfo.radius1, co);
        Sincos (thetai, si, ci);
        FCMultiply (si, state↑.stateinfo.radius2, si);
        FCMultiply (ci, state↑.stateinfo.radius2, ci);
        outmsg := NewQVector (Qvec2);
        FCAdd (ci, co, outmsg↑.v4[0]);
        FCAdd (si, so, outmsg↑.v4[1]);
        SendMsg (outmsg, 2);
    
```

```

{-----}
{ Test whether the figure is complete. }
{-----}

outmsg := NewQBoolean;
outmsg↑.b := TRUE;
IF (state↑.stateinfo.currenti = 0) AND
    (state↑.stateinfo.currento = 0) THEN BEGIN
    newdata := TRUE;
    SendMsg (outmsg, 4);
    END
ELSE
    endMsg (outmsg, 3);
END;
IF Cleaninputs THEN
    inputs := ckinputs (1, 4)
ELSE inputs := NIL;
END;
END.    { GenFunction }

```

6.5 CONCLUSION

This concludes the formal instructions for writing your own user-written functions. Once you are familiar with the processes and examples described in section AP5 thru section AP8 should be most helpful in providing a quick source of information on the utility routines and other information you will need to write your own functions.

Section AP7 contains instructions for transferring S-record files from the host system to the PS 390 firmware diskette, as well as instructions for protecting user-written functions from PS 390 global INIT commands. Once user-written functions reside on the firmware diskette, they will load in approximately the same manner as intrinsic PS 390 functions. Section AP7 also contains instructions on how to use the PS 390 Debugger.

Section AP7

Loading and Debugging UWFs

Section 7.1 describes how to load a user-written function (UWF) from a runtime diskette (and optionally create an instance of the function) when the PS 390 is booted, in such a way that both the function code and the function instance are protected from INITIALIZE commands. For example, this facility might be used if you have written a function to control a peripheral device, such as a mouse, and you want to use the function in the same way as the ordinary PS 390 initial function instances.

Section 7.2 contains the various commands for the PS 390 Debugger and explanations of how to use them to determine whether a section of code for a user-written function is actually being executed or not.

7.1 Loading User-Written Functions From Diskette

Before proceeding with this section, you should know how to transfer files from the host computer to a diskette using the UTILITY program available on PS 390 Diagnostic Utility Diskette. You should be familiar with name suffixing conventions and how to use Configure mode on the PS 390. You may also find it useful to refer to the block diagrams for the CONFIG.DAT file. All of this information can be found in RM7 of the *PS 390 Document Set*.

To load a user-written function from the firmware diskette, the file containing the user-written function must first be downloaded to the diskette from the host. This file should contain the function header line followed by the S-record output from the linker, and should terminate with a semicolon. It should not contain any multiplexing bytes. To download the file to the diskette, follow the instructions for using the UTILITY program in RM12 of the *PS 390 Document Set*. The file should be transferred as an ASCII file and given an extension of .DAT on the PS 390 diskette.

Unless you are using a PS 390 with two disk drives, the procedures described in the following sections require that the file containing the function code be on the same diskette as the SITE.DAT file.

To load the user-written function file on the diskette, you must modify the SITE.DAT file to include a function network which will read the file from the disk and route its contents to the function which will load the user-written function into memory.

If you want to instance the user-written function at boot time, you cannot just include the PS 390 commands to do so in SITE.DAT; this is because SITE.DAT is processed before the user-written function has been read in and loaded into memory. Instead, you must put the commands necessary to instance and initialize the user-written function in a separate file and modify SITE.DAT to include a network to read in this file after the user-written function code has been loaded.

The function and its instances can be protected from INITIALIZE commands by creating them using a different CI (command interpreter) than that used for commands received from the host or the keyboard. (Remember that an INITIALIZE command removes only those names which were created by the CI receiving the command.) You will use the CI numbered 0, which is also used for setting up the ordinary initial function instances from commands read from CONFIG.DAT.

7.1.1 Loading the User-Written Function Into Mass Memory

The following version of SITE.DAT sets up a function network which loads a user-written function from the file EXAMPLE.DAT into memory. After creating the file on the host, the PS 390 UTILITY program should be used to transfer it to the PS 390 diskette.

```

configure sezme;
dcwait1 := f:timeout;           { to force delay before reading UWF }
StartUWF1 := f:constant;        { holds name of file containing UWF }
LoadUWF1 := f:readdisk;         { function to read the file }
send fix(500) to <2>dcwait1;     { cause 5-second delay }
conn dcwait1<2> : <1>StartUWF1;
send 'EXAMPLE' to <2>StartUWF1; { then send filename to read function }
conn StartUWF1<1> : <1>LoadUWF1;
conn LoadUWF1<1> : <1>srec_gather0; { route file contents to UWF loader }
send fix(0) to <2>srec_gather0;  { CI number to associate with UWF }
disconn srec_gather0<1> : all;   { normally connected to a CIROUTE }
disconn srec_gather0<3> : all;
send true to <1>dcwait1;        { kick the thing to get it started }
finish configuration;

```

This is the bare minimum required. Note that since the SITE.DAT file is read in Configure mode, you have to be sure to include the proper suffixes on all the names referenced.

The function `dwait1` is used to force a 5-second delay between processing the SITE.DAT file and reading the file containing the user-written function. This delay is necessary with the PS 350 data concentrator initialization sequence but is not necessary with the PS 390 peripheral multiplexer. This initialization takes place immediately after SITE.DAT has been read. To avoid conflicts, it is important to allow sufficient time for the initialization to complete before trying to read from the diskette.

After `dwait1` has been triggered and the delay time elapsed, it will send a message to input <1> of `StartUWF1`. In turn, this will send the name of the file to `LoadUWF1`, which reads the file from the diskette. The contents of the file are routed to `srec_gather0`, an instance of `F:GATHER_GENFCN`.

Sending a value of `fix(0)` to input <2> of `srec_gather0` associates the name of all the user-written functions created by `srec_gather0` with CI number 0. This means that the names of these functions are protected from an `INITIALIZE` command on any other CI. Note that this does not protect instances of these functions from `INITIALIZE` commands. Also, you must still be careful not to redefine the name of the function; i.e.,

```
example := f:example;
```

will still destroy the function body.

7.1.2 Loading the User-Written Function and Creating an Instance

If you want to create an instance of a user-written function at boot time, you should put the PS 390 commands necessary to do so in a separate file, which might be called `SETUP.DAT`. You can then modify `SITE.DAT` to include a network that reads `SETUP.DAT` from the diskette and sends its contents to the same CI that processes the `CONFIG.DAT` and `SITE.DAT` files. The important thing to remember is that you cannot instance a user-written function until it has been loaded, so `SETUP.DAT` cannot be read in until the file containing the user-written function has been read in.

There are a few restrictions on what `SETUP.DAT` can contain. First of all, the CI used to process this file does not handle implicit name suffixing properly. You should always use Configure mode in `SETUP.DAT`, and be

careful to include explicitly the proper suffixes on all names. Secondly, you cannot DISPLAY anything through this CI.

The contents of SETUP.DAT will vary depending on the application. Here is an example:

```
configure sezme;
myexample1 := f:example1;          { whatever needs to be done to a initialize the }
                                   { function                                }

setup cness true <2>myexample1;
send true to <2>myexample1;
send 'System is ready for use' & char(13) & char(10) to <1>es_te1;
finish configuration;
```

Note that, in Configure mode, you have to suffix the name of the user-written function, as well as any function instances that you refer to.

Once you have created the SETUP.DAT file on the host, use the PS 390 UTILITY program to transfer the file to the PS 390 diskette in the usual way. Unless you have a two-drive system, this file must be placed on the same diskette as the SITE.DAT file.

Here is the SITE.DAT file to read the function code from EXAMPLE.DAT and the commands from SETUP.DAT. This network is illustrated in Diagram 2.

```
configure sezme;
dcwait1 := f:timeout;              { to force delay before reading UWF }
delay1  := f:timeout;              { to delay before reading SETUP.DAT }
StartUWF1 := f:constant;           { holds name of file containing UWF }
LoadUWF1 := f:readdisk;            { function to read the file }
StartSetup1 := f:constant;
LoadSetup1 := f:readdisk;

send fix(500) to <2>dcwait1;        { cause 5-second delay }
conn dcwait1<2> : <1>StartUWF1;
send 'EXAMPLE' to <2>StartUWF1;     { then send filename to read function }
conn StartUWF1<1> : <1>LoadUWF1;
conn LoadUWF1<1> : <1>srec_gather0; { route file contents to UWF loader }
send fix(0) to <2>srec_gather0;     { CI number to associate with UWF }
disconn srec_gather0<1> : all;      { normally connected to a CIROUTE }
disconn srec_gather0<3> : all;
conn LoadUWF1<2> : <1>delay1;       { trigger when UWF file is read }
send fix(100) to <2>delay1;         { 1-second delay }
conn delay1<2> : <1>StartSetup1;    { then fire the function to read SETUP }
```



```

send 'SETUP' to <2>StartSetup1;
conn StartSetup1<1> : <1>LoadSetup1;
conn LoadSetup1<1> : <1>rfchop$;                                     { to send it to the CI }
send true to <1>dcwait1;                                           { kick the thing to get it started }
finish configuration;

```

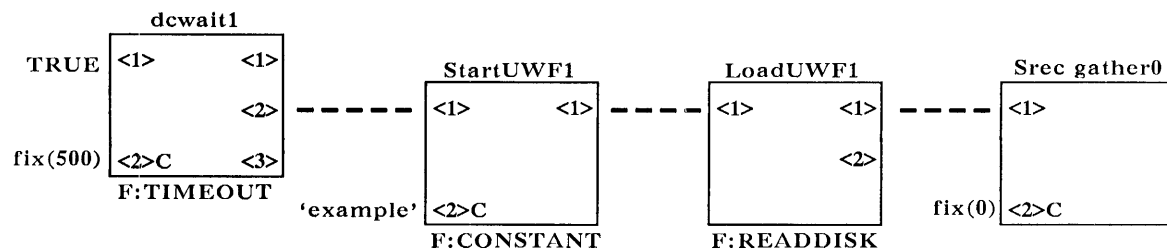
The first part of this network is the same as in the previous example. Output <2> of LoadUWF1 is used to signal when the file containing the code for the user-written function has finished being read in. Use this message to trigger reading SETUP.DAT, after a one-second delay. (Experience has shown that this delay is necessary.) The contents of SETUP.DAT are routed to rfchop\$.

7.1.3 CONCLUSION

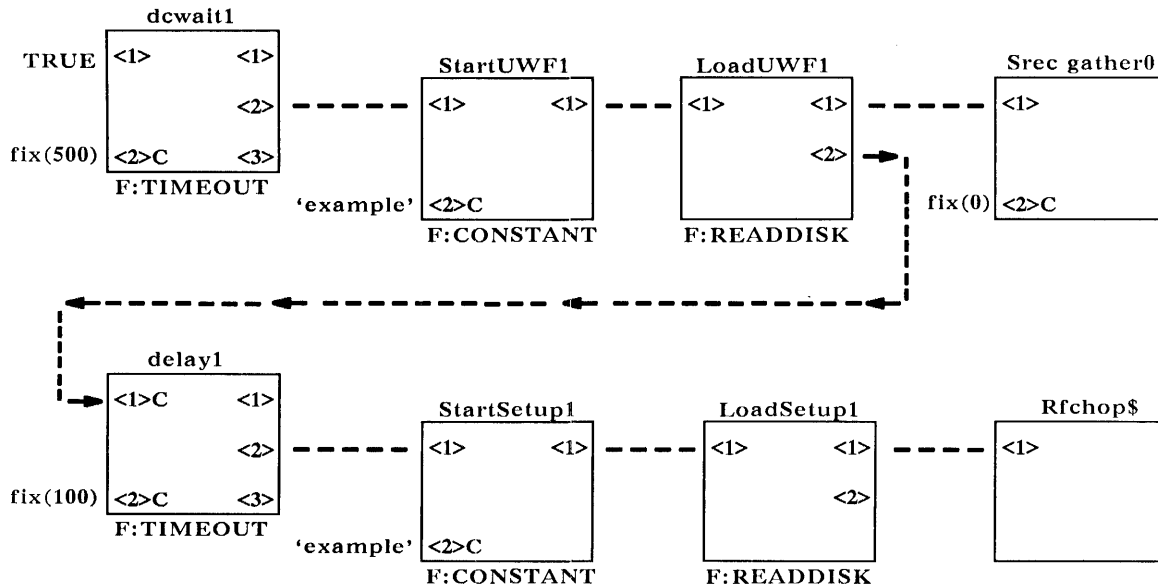
The diagrams on the next page illustrate the function networks set up by the sample SITE.DAT files used in this section.

This concludes the instructions for transferring S-records to the firmware diskette and initializing them at boot time.

Function Network Diagram 1



Function Network Diagram 2



7.2 PS 390 Debugger

The PS 390 Debugger (Debug) can sometimes aid in debugging a user-written function. However, Debug is rather primitive and the procedure for locating the code for a user-written function in mass memory is complicated. It is suggested that you do not attempt to use Debug except when other methods for debugging a function have failed. You must be familiar with assembly language and the Motorola listing file formats to understand what is required to use Debug. This section describes:

- How to use Debug.
- The Debug commands.
- How to set breakpoints in your user-written function. Setting breakpoints is useful for determining whether a section of code in your function is actually being executed or not.

7.2.1 Using the Debugger

To use the debugger, an ASCII terminal must be attached to PS 390 Port 3. The serial port used by Debug is initialized to 8 bits, no parity, and one stop bit; each byte is stripped to 7 bits in case the terminal being used sets a parity bit, and the baud rate is set to 9600. Should you want to modify these

characteristics, you can do so by using the SETUP INTERFACE command for port30.

Debug mode can be entered by pressing the BREAK key on the ASCII terminal. When Debug is entered, an asterisk "*" is displayed on the terminal and the PS 390 display is blanked.

The asterisk is a prompt character that appears whenever Debug is expecting a command to be entered. All Debug commands are either one or two characters in length. Debug converts all lowercase characters to uppercase automatically. Whenever an invalid command is entered, Debug outputs a BEL and "?" character, moves to the next line, and prompts for a new command. Refer to the next section for tables containing the Debug commands.

Because of its requirement to run on a minimum amount of hardware, Debug has several limitations. First, minimal editing of input is allowed. Second, all commands execute immediately upon pressing the appropriate key. Third, all numbers used by Debug must be hexadecimal (base 16) rather than decimal (base 10).

Any time a number is required by a Debug command, a 32-bit register is cleared to zero, and each digit that is entered into the register is shifted in from the right. If more than eight digits are entered, the upper digit is shifted out the left end of the register and lost. Numbers are considered completely entered when the first non-hexadecimal character is entered. (This implies that if the first character entered is not hexadecimal, the numeric value is zero.)

Any time that a hex number may be entered, one of the following characters may be entered to provide a different meaning. Note that no additional delimiter character is required as with the entry of hexadecimal numbers.

- Specify an ASCII string. The string is terminated by a second single quote. All characters are taken exactly as typed (no lower case to upper case conversion) including control characters such as carriage return and line feed (CTRL Y will still kill the command, however). Note that there is no way to insert a single quote character in this mode. This should be most useful with the hunt commands (HB, HW, HL) or for inserting single characters into memory. Example: "I 'a'" would insert the value X'61' into the current open location.

- O** Use the current open location as the hex value. Most useful with the list command (L) or to set breakpoints (BR). Example: "L O ." would list from the current open location for one line.
- P** Use the current program counter value as the hex value. Most useful with the list command (L), to set breakpoints (BR), or with the open command (O). Example: "BR P" would set a break point at the current program counter location.
- S** Use the current stack pointer value as the hex value. Most useful with the list command (L) or with the open command (O). Example: "O S" would open the location pointed to by the stack pointer.
- +** Use the current open location plus the specified offset. For example: "O +10." would open the location 16 (X'10') bytes beyond the current open location (note that a delimiter is needed here). Or, "L +10, ." would list the line 16 bytes beyond the current open location.
- Use the current open location minus the specified offset. For example: "O -A." would open the location 10 (X'A') bytes before the current open location (note that a delimiter is needed here).

When entering a hex number, you may now use either backspace or delete to correct the number. Up to the last 8 digits may be deleted from the number. A digit is deleted each time the delete or backspace key is pressed until the number being built is found to be all zero. Note that if more than 8 digits are entered, the first digits will have shifted out of the register and will no longer exist (but they will still appear on the screen). With this new feature, however, there should be no need to enter numbers longer than 8 digits in the first place. Note that backspace and delete do not work for any ASCII strings.

Three special characters control output of data to the terminal:

<CTRL>S

Temporarily stops the sending of output to the terminal.

<CTRL>Q

Resumes the sending of output to the terminal after a <CTRL> S. (Note that program code occasionally may miss a <CTRL> S<CTRL> Q sequence when displaying great amounts of data at a baud rate over 2400.)

<CTRL>Y

Permanently stops the sending of output to the terminal. <CTRL> Y may also be used to terminate any command before the last character of the command is entered.

Any unrecognized character is echoed to the terminal but is otherwise ignored. Most valid commands move the cursor on the terminal to the next line to indicate successful completion of the command.

7.2.2 Debugger Commands

The following table lists the valid Debug commands. Most of the information in the tables is provided for completeness; only a few of these commands will be needed while using the debugger to set breakpoints in your code. The Debug program keeps a pointer to a specific location currently in use. This location is referred to as the “open Location.” All commands in the following table are associated with the open location. If a location has not been opened after a Reset or after stopping program execution using the O or Q commands, the current open location may be invalid and could cause a bus error if accessed.

Table 7.2-1 Commands Accessing “Open” Memory Locations

COMMAND	DESCRIPTION
O	Open a location and display its contents. For example, to open location X'BA3E': <div>OBA3E. (Entered like this. Any delimiter can be used in place of the period in this and in the following examples.)</div> <div>* O BA3E. (Displayed like this.) 0000BA3E 00 *</div>
Q	Open a location but do not display its contents. For example, to open location X'FFF819':

(continued on next page)

Table 7.2-1 Commands Accessing “Open” Memory Locations (continued)

COMMAND	DESCRIPTION
	<p>QFFF819. (Entered like this.)</p> <p>*Q FFF819. (Displayed like this. Note that the * * cursor moves to a new line but nothing else is displayed.)</p>
SPACE	<p>Insert a byte of data into the current open location and then open and display the contents of the next location. For example, if the current open location is X'BA3E' and the programmer desires to insert the bytes X'60' and X'FE' in this and the next location:</p> <p>_60._FE. (Entered like this. In this example, “_” means space.)</p> <p>* 60. (Displayed like this.)</p> <p>0000BA3F 00 * FE.</p> <p>0000BA40 00 *</p>
I	<p>Insert a byte of data into the current open location but do not move on to the next location and do not display the contents of any location. For example, if the current open location is X'FFF819' and the programmer desires to insert the value X'00' into the location:</p> <p>I51. (Entered like this.)</p> <p>*I 51. (Displayed like this.)</p> <p>*</p>
W	<p>Insert a word of data at the current open location. This command should be used when writing data to a register that is addressable on a word basis only. If the current open location is an odd address, the word is inserted into the next lower even address (to avoid an address error). For example, if the current open location is X'321FE' and the operator wants to insert the word X'1234' into the location:</p> <p>W1234. (Entered like this.)</p> <p>*W 1234 (displayed like this.)</p> <p>*</p>

(continued on next page)

Table 7.2-1 Commands Accessing “Open” Memory Locations (continued)

COMMAND	DESCRIPTION
.	<p>Display the address and contents of the current open location. For example, if the current open location is X'FFF819':</p> <p>. (Entered like this.) *. (Displayed like this.) 0FFF819 51 *</p>
+	<p>Open the location after the current open location and display its contents. For example, if the current open location is X'BA3E':</p> <p>+ (Entered like this.) *+ (Displayed like this.) 0000BA3F FE *</p>
/	<p>Open the location after the current open location and display its contents. Identical to “+” but does not require the use of the shift key.</p>
-	<p>Open the location before the current open location and display its contents. For example, if the current open location is X'BA3F', this command would work as follows:</p> <p>- (Entered like this.) *- (Displayed like this.) 0000BA3E 60 *</p>
^	<p>The pointer following command. This command takes the 32-bit value at the open location and uses it as the new open location. Note that it does not check that the current open location is an even location (odd causes an address error), or that the value at the open location is a valid memory address (invalid causes a bus error). The following is an example of how to use this and other new commands to find the front of a diagnostic program and set a breakpoint at a subroutine listed as being at X'7DC' in the link map:</p> <p>R Display registers (assume A5 = 3A23E). O 3A23E. Open base of global variables.</p>

(continued on next page)

Table 7.2-1 Commands Accessing “Open” Memory Locations (continued)

COMMAND	DESCRIPTION
O -A.	Get to diagnostic base address.
^	Open the first location of the diagnostic.
BR +7DC.	Set a breakpoint at the beginning of the subroutine.
CTRL G	Begin execution at the current open location. Do not set up any of the registers before beginning and do not enable breakpoints. This should normally be used to begin program execution when local memory may not be used to hold the exception vectors. Nothing is displayed by this command.

Table 7.2-2 Commands to List Data in Memory

COMMAND	DESCRIPTION
L	<p>List a block of data in both hexadecimal representation and ASCII representation. All unprintable characters are displayed as a period. For example, to display locations X'40000' to X'4002F':</p> <p>L40000,4002F. (Entered like this; period and comma are arbitrary delimiters.)</p> <p>(Displayed like this.)</p> <pre>L40000, 4002F. 00040000 00 01 00 00 00 04 00 14 00 01 00 00 00 04 02 DAZ 00040010 60 00 0A EC 10 38 F8 57 11 FC 00 4E F8 55 11 FC ...1.8xW...NxU.. 00040020 00 7A F8 55 11 FC 00 37 F8 F7 11 FC 00 41 F8 51 .zxU...7xW...AxQ</pre> <p>At this point, the current open location is set to X'40000'. For convenience, if the ending address is less than the starting address, the ending address is considered to be a byte-count. The list command accepts input in the form of:</p> <p style="padding-left: 40px;">L<starting-address>,<number-of-bytes>.</p> <p>as well as:</p> <p style="padding-left: 40px;">L<starting-address>,<ending-address>.</p> <p>For example, the following command could be used to display locations X'40000' to X'4002F':</p> <p style="padding-left: 40px;">L40000,2F.</p> <p>These addresses are obtained by a common routine which converts odd numbers to even numbers (so commands like "HW" do not get an address error). If an odd number is entered as the starting address, the first byte listed is actually the byte at the previous even address.</p> <p>> Display the line just after the last line displayed by the list (L) command. This is most useful when the block of memory listed was almost big enough but not quite.</p> <p>< Display the line just before the last line displayed by the list (L) command. This is most useful when one line of data was displayed (such as the current program counter location, and it is desired to see what was immediately in front of it.</p>

Table 7.2-3 Program Execution and Debugging Commands

COMMAND	DESCRIPTION
R	<p>Display contents of all processor registers at the time of the last exception. These contents are the values utilized by the "T" or "G" command. The values may be modified using the "A", "D", "P", "S", or "U" command. Most of these values are initialized to 0 by the "V" command. Registers are displayed in the following format:</p> <pre> *R PC = 00000000 SR = 0000 USP = 00000000 D0-D7= 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 A0-A7= 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 </pre>
P	<p>Set the program counter to a new value. For example, to set the program counter to the value X'40014':</p> <pre> P40014. (Entered like this.) P 40014. (Displayed like this; note that the cursor moves to a new line but nothing else is displayed.) </pre>
S	<p>Insert a new value into the status register. For example, to set the status register to the value X'2700':</p> <pre> S2700. (Entered like this.) S 2700. (Displayed like this; note that the cursor moves to a new line, but nothing else is displayed.) </pre>
U	<p>Insert a new value into the user stack pointer (USP). For example to set the user stack pointer to the value X'36972':</p> <pre> U36972. (Entered like this.) U 36972. (Displayed like this; note that the cursor moves to a new line, but nothing else is displayed.) </pre>
D0-D7	<p>Change the value of the specified address register. For example, to set address register D2 to X'12345678':</p>

(continued on next page)

Table 7.2-3 Program Execution and Debugging Commands (continued)

COMMAND	DESCRIPTION
	<p>D212345678. (Entered like-this.)</p> <p>D2 12345678. (Displayed like this.)</p>
A0-A7	<p>Change the value of the specified address register. For example, to set address register A4 to X'FFFFFF850':</p> <p>A4FFFFFF850. (Entered like this.)</p> <p>A4 FFFFFFF850. (Displayed like this.)</p>
G	<p>Begin program execution with all registers set up as displayed by the "R" command. Immediately before execution, all breakpoints are initialized in memory. If the first instruction to be executed is at a breakpoint address, several microseconds pass between the execution of the first instruction and any following instructions. If the program counter is odd, execution begins at the next lower even address. Nothing is output to the screen to indicate that the "G" command has been executed.</p>
T	<p>Trace one instruction. All registers are set up as displayed by the "R" command. This instruction uses the trace bit in the MC68000 microprocessor. Since breakpoints actually are only set up in memory when the "G" command is executed, they have no effect on the trace command. If the program counter is odd, execution begins at the next lower even address. After tracing one instruction, the following is displayed (with actual register values filled in):</p>
*T	<p>Trace PC = 00000000 SR = 0000 USP = 00000000</p> <p>D0-D7= 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000</p> <p>A0-A7= 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000</p>
V	<p>Initialize exception vectors, registers, stack pointer, and breakpoint table. Care should be taken not to type a "V" while debugging your program, as the machine state will be lost!</p>

Up to seven breakpoints may be set for Debug programs. Debug sets a breakpoint by storing a TRAP #15 instruction at the breakpoint location when the “G” command is executed. Every time an exception occurs that causes DEBUG to be entered (e.g. pressing the BREAK key or encountering a breakpoint), the current open location is set equal to the program counter address. This is especially useful when using the trace command or when using breakpoints.

The programmer must assure that no attempt is made to set a breakpoint at a nonexistent location. If the breakpoint is set at an odd location, the next lower even address is used. It is important that the breakpoint be set at the beginning of an instruction rather than in the instruction parameter part.

If there is not enough room to set another breakpoint, Debug indicates it is full with the following message:

Break table is full

Table 7.2-4 Breakpoint-Related Commands

COMMAND	DESCRIPTION
BR	Set one breakpoint. A maximum of 7 breakpoints may be set up at any one time. If this number is exceeded, the following message is displayed: Break table is full For example, to set a breakpoint at location X'C000': BRC000. (Entered like this.) *BR C000. (Displayed like this.)
BD	Display all breakpoints. For example, if three breakpoints have been set, the following might be displayed: BD (Entered like this) Breakpoints = 0000C000 0000CAB8 0000C124 (Displayed like this)

(continued on next page)

Table 7.2-4 Breakpoint-Related Commands (continued)

COMMAND	DESCRIPTION
If no breakpoints are set, the following message is displayed:	
Breakpoints =	
BC	Clear one breakpoint. If the system is unable to clear the breakpoint for any reason, the cursor remains on the same line. If the breakpoint is successfully cleared, a carriage return is output to indicate successful completion. If the breakpoint is not successfully cleared, the audible alarm sounds. For example, to clear a breakpoint that was set at location X'C000': BCC000. (Entered like this.) BC C000. (Displayed like this.)
BA	Remove all breakpoints.

The Hunt commands HW, HB, and HL are used to look for a word (16 bits), for a byte (8 bits), or for a long word (32 bits), respectively, in a search range designated by the programmer.

Table 7.2-5 Hunt Commands

COMMAND	DESCRIPTION
HW	Hunt for a word (16 bit) pattern within a specified search range designated by the programmer: <div style="display: flex; justify-content: space-between; margin-left: 150px;"> HW400,10000. (Entered like this.) </div> <div style="display: flex; justify-content: space-between; margin-left: 150px;"> HW 400, 10000. (Displayed like this.) </div>

NOTE

If the specified search range is to be X'100' to X'1FF', enter the number range 100 to 200. The ending address is the one immediately preceding the address entered by the operator.

A prompt then asks for the pattern to search for, and the programmer responds as in the following example:

Pattern to search for: 4ED0

Once a matching pattern is found, its address is displayed, along with the 16 bytes following the address, as in the following example:

0000EB64 4E D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Pressing any key on the terminal continues the search for another matching pattern from the point after the last pattern found until the end of the range. A <CTRL> Y may be used to stop the search at any time. When no matching pattern is found, the following message is displayed:

Not found

If the pattern to search for is an ASCII sequence of characters, the characters must be surrounded by single quote marks. Note that if an entry is found within 16 bytes of an end address, the whole line is not printed to avoid a bus error in case the end address is on a memory boundary.

(continued on next page)

Table 7.2-5 Hunt Commands (continued)

COMMAND	DESCRIPTION
HB	Hunt for a byte (8 bit) pattern within a specified search range designated by the programmer.
HL	Hunt for a long word (32 bit) pattern within a specified search range designated by the programmer.

Table 7.2-6 Boot-related Commands

COMMAND	DESCRIPTION
---------	-------------

NOTE

The error-correction bits of memory must be initialized by either the confidence tests or a memory test (such as M1) prior to execution of the boot-related commands.

Should an error occur while executing one of the boot-related commands, additional error information may be obtained by reading the diskette controller status bits at location X'FFF811'.

- BO** Load and execute the boot file from the minifloppy. This command must be followed immediately by a carriage return.
- A "V" command is automatically executed immediately before accessing the diskette controller to guarantee that memory error correction logic is enabled and that the exception vectors are initialized.
- The floppy disk drive is turned off after the file is loaded into memory. Any code that uses the disk immediately after being loaded must explicitly turn the disk motor on again.

NOTE

All breakpoints disappear when the boot command is executed.

If an error is encountered, one on of the following messages is displayed:

Disk initialization error

This error message indicates that either the diskette drive does not respond properly or that there is no diskette in the drive:

Error in locating boot file

This error message indicates that either the diskette could not be read or that the diskette does not contain a valid boot file.

(continued on next page)

Table 7.2-6 Boot-related Commands (continued)

COMMAND	DESCRIPTION
	<p>Error in loading boot file</p> <p>This error message indicates that either a read or a seek error occurred while reading in the boot file.</p>
BN	<p>Load the boot file in from the diskette but do not begin execution. This command is identical to the BO command except that when the file is loaded, instead of beginning execution immediately, control is returned to the programmer and the following message is displayed:</p> <p>Start address = XXXXXXXX</p> <p>where XXXXXXXX is the address where the program was loaded. This command must be followed immediately by a carriage return.</p> <p>Note that a “V” command is automatically executed immediately before accessing the diskette controller to guarantee that memory error correction logic is enabled and that the exception vectors are initialized. This means that all breakpoints disappear when the boot command is executed.</p> <p>The floppy disk drive is turned off after the file is loaded into memory. Any code that uses the disk immediately after being loaded must explicitly turn the disk motor on again. After the file is loaded, the start address is opened. A CTRL G command may be entered without explicitly opening the start address.</p>
BX	<p>Load the specified file from the diskette to the specified address. This command must be followed immediately by a carriage return. The file name is entered following the prompt message:</p> <p>Enter name of file:</p> <p>The name may be from 1 to 8 characters in length. Should an error be made in entering the file name, type a CTRL Y and begin again.</p> <p>No local memory accesses of any kind occur (unless the load address is within local memory). If any breakpoints are set at the time that the “BX” command is entered, the original instructions at the breakpoint locations may not be restored.</p> <p>(continued on next page)</p>

Table 7.2-6 Boot-related Commands (continued)

COMMAND	DESCRIPTION
---------	-------------

NOTE

Debug does not allow the programmer to specify the extension or file version number. The file must be of the type .EXS (executable stand-alone) to be loadable by the BX command. The highest version number of the file is loaded in automatically; there is no way to select an alternate version number.

The load address is entered following the prompt message:

Enter load address:

If the address is odd, the next lower even address is used. This eliminates the possibility of an address error while executing the specified file.

The floppy disk drive is turned off after the file is loaded into memory. Any code that uses the disk immediately after being loaded must explicitly turn the disk motor on again. After the file is loaded, the start address is opened. A CTRL G command may be entered without explicitly opening the start address.

Table 7.2-7 Memory Test Commands

COMMAND	DESCRIPTION
<p style="text-align: center;">NOTE</p> <p>Memory tests M3 through M6 strobe the DIAGSYNC line on the GCP card whenever a memory error is detected. This allows a logic analyzer to capture the state of the hardware at the time of the memory error.</p>	
M1	<p>Test the GCP local memory. This test starts at location X'0008' and tests all of local memory. It performs a stripes pattern test of all bits for both 1's and 0's, coming from both directions.</p> <p>After all normal memory bits have been tested, the test above is repeated using the check bits. This test has no parameters. Errors are displayed in the following manner:</p> <p style="text-align: center;">00000100 Expected-5555 Received-5554 Bits in error-0001</p> <p>where the first field is the address where the error was detected, the second is the expected value, the third is the received value, and the fourth contains the bits in error as determined by the test.</p>
M2	Test the error correction and detection circuitry of the local memory.
M3	<p>Perform a simple and quick complement test. M3 reads a location, complements it, and then verifies that all bits have changed. For example, to test memory from X'0008' to X'03FF':</p> <div style="display: flex; justify-content: space-between;"> <div>M38,400.</div> <div>(Entered in this form.)</div> </div> <div style="display: flex; justify-content: space-between;"> <div>M3 8, 400.</div> <div>(Displayed in this form.)</div> </div> <p>Note that the last location tested is X'3FE' (word address), which includes location X'3FF' (byte address). The end address corresponds to the first even address after the last address to be tested. A carriage return is output when the test is started, and another carriage return is output when the test has been completed successfully. Errors are displayed in the same way as in test M1.</p>

(continued on next page)

Table 7.2-7 Memory Test Commands (continued)

COMMAND	DESCRIPTION				
<p style="text-align: center;">NOTE</p> <p>For convenience, if the ending address is less than the starting address, the ending address is considered to be a byte count. The M3 command accepts input in the form of:</p> <p style="text-align: center;">M3<starting-address>,<number-of-bytes>.</p> <p>as well as:</p> <p style="text-align: center;">M3<starting-address>,<ending-address>.</p> <p>For example, the following command could be used to test locations X'200000' to X'2001FE':</p> <table><tr><td>M3200000,200.</td><td>(Entered in this form)</td></tr><tr><td>M3 200000, 200.</td><td>(Displayed in this form)</td></tr></table> <p>These addresses are obtained by a common routine which converts odd numbers to even numbers. This eliminates the possibility of an address error.</p>		M3200000,200.	(Entered in this form)	M3 200000, 200.	(Displayed in this form)
M3200000,200.	(Entered in this form)				
M3 200000, 200.	(Displayed in this form)				
M4	Perform an address line test. It stores the low order 16 bits of the address at the test address and then, when all of memory to be tested has been filled, it verifies that the proper data has been stored. Addresses are specified the same way as in test M3.				
MA	Execute the read and verify portion of the memory address lines test (M4). This command is intended to be preceded by an M4 test over the same range (and thus verifies that the data written by M4 is still intact). Following is an example of how it might be used:				

(continued on next page)

Table 7.2-7 Memory Test Commands (continued)

COMMAND	DESCRIPTION
	M4 200000, 210000. (Write the data pattern) <wait a while> MA 200000, 210000. (Verify data integrity)
M5	Perform a simple stripes test. The test procedures are the same as those described in memory test command M1 above, except the operator specifies the test range and the MMR (Mass Memory Maintenance Register) is not modified.
M6	Store a random pattern throughout the test memory, using a pseudo-random-number generator. After all of memory has been filled, the seed of the random number generator is reset and then all memory locations are verified. Addresses are specified in the same way as with test M3.

7.3 Setting Breakpoints in Your Code

This section describes the steps you must follow to set a breakpoint in your code. Where appropriate, examples are given. The user-written function F:MAG will be used for the example. It is assumed that you want to put a breakpoint right after the function has checked its input queues upon being executed.

1. Make sure that your compile, assemble and links are done with the List option set on. (Refer to the Motorola cross-software manual for more details on the List option.) This causes creation of listing and link-map files which are necessary to find the location to a set breakpoint. For F:MAG, the commands to do this on VAX/VMS are:

```
$ xpas mag
$ xpas2 mag;L
$ xlink mag/userlink,mag,mag;himx
```

2. Find the offset within the user-written function code of the breakpoint location using the listing file from pass 2 of the compiler and the link map.
 - a. Find the address within the link of the beginning of the file containing your code. The link map will give you this information. In the following example, for F:MAG, the code for the user-written function (Module USERFUN) starts at location 2CC.

Load Map:

```
Segment SEG0: 00000000 000000FF 0,1,2,3,4,5,6,7
Module      S  T  Start      End      Externally Defined Symbols

USERLINK 1 00000000 00000019
USERLINK 6 0000001A 00000025

Segment SEG1(R): 00000100 000005FF 8,9,10,11,12,13,14
Module      S  T  Start      End      Externally Defined Symbols

USERLINK 8 00000100 00000173 .PALSTS      00000108 .PDIS      00000156
               .PLJSR      00000100 .PNEW      0000012E
```

USERLINK 9	00000174	000002CB	FRAMES	0000027C	SENDMSG	0000018C
			SET_CNES	000002C4	SYSTEMER	000001B6
			TEXT_TEX	00000294	UWFERROR	000002BC
			HRTIME	00000284	QILLMESS	00000198
			QILLVALU	0000019E	QINCOMPA	000001A4
			FCADD	000001C8	FCDIVIDE	000001F4
			FCINT2DO	000001FA	FCINTEGE	00000206
			FCMULTIP	000001D4	FCNEARZE	0000020C
			FCP2MULT	000001EE	FCROUND	00000200
			FCSQROOT	00000212	FCSUBTRA	000001CE
			SINCOS	00000218	VFETCH	0000024E
			TICKS	0000026C	TIME_TEX	000002A4
			CHAR_TEX	0000028C	MSGCOPY	000001AA
			RNDMNUMB	0000021E	CKINPUTS	00000180
			CKPRIVAT	00000174	CLEANINP	00000186
			QSEND COP	00000192	MY_IN_OU	00000264
			MY_NAME	0000025C	NEWQBOOL	00000236
			NEWQINTE	00000230	NEWQMATR	00000248
			NEWQNIL	0000023C	NEWQPACK	00000224
			NEWQREAL	0000022A	NEWQVECT	00000242
			NEWTRY	000002B4	SAVEPRIV	0000017A
			CSECS	00000274	DROPMESS	000001B0
			FPABS	000001C2	FPECOMP	000001BC
			VSTORE	00000254	INT_TEXT	0000029C
			REAL_TEX	000002AC		
USERFUN 9	000002CC	0000057F	GENFUNCT	000002CC		

- b. Find the offset within your file of the particular instruction you want to set the breakpoint at. The listing file from pass 2 of the compiler will give you this information.

In this example, the breakpoint will be set just after the initial call to CkInputs. Here is the relevant part of the listing file:

```

*. PROCEDURE GenFunction ;                               326
*.                                                         327
*. VAR                                                     328
*.   inputs : PtrUWFInQarray;                             329
*.   outmsg : Ptrqdata;                                   330
*.   temp   : Double;                                     331
*.                                                         332
*. BEGIN { GenFunction }                                  333
*.   inputs := CkInputs (1, 1);                             334
00000000                                USER50 EQU      *
00000000 2F2D 000C                                MOVE.L 12(A5), -(A7)

```

00000004 4E56 FFFF	LINK A6,#-L1	
00000008 2B4E 000C	MOVE.L A6,12(A5)	
0000000C 598F	SUB.L #4,A7	
0000000E 7001	MOVE.L #1,D0	
00000010 3F00	MOVE D0,-(A7)	
00000012 7001	MOVE.L #1,D0	
00000014 3F00	MOVE D0,-(A7)	
00000016 4E93	JSR (A3) CUP	
00000018	DC.L \$00FFFFE8	
0000001C 2D5F FFFC	MOVE.L (A7)+,-4(A6)	
*. WHILE inputs <> NIL DO BEGIN		335
00000020	L2 EQU *	
00000020 206E FFFC	MOVE.L -4(A6),A0 ---->(see note)	
00000024 227C 00000000	MOVE.L #0,A1	
0000002A B1C9	CMP.L A1,A0	
0000002C 6700 0000	BEQ L3	

NOTE

The location where you should set the breakpoint is offset 0020 from the beginning of the module.

- c. Add the two numbers arrived at in (a) and (b). This will give you the actual offset from the beginning of the user-written function of the instruction at which to set the breakpoint. In this example, the actual offset is $2CC + 20 = 2EC$. (Remember, all numbers are Hex.)
3. Find the address in PS 390 mass memory of the start of the function code. After downloading the user-written function so that it is in place in the PS 390 mass memory, enter Debug by depressing the Break key on the ASCII terminal connected to port 3. Look through mass memory (using the HL or HW command) for the name of the function. Use the first 4 characters of the name only. Remember that all names have a suffix appended to the end of the name. For most functions downloaded from the host the suffix is a "1" (one). Hunt through memory starting at location 200000 and ending at 300000 (or 400000, if you have 2 megabytes of mass memory, 500000, if you have 3 megabytes, etc). The example function is named MAG. Thus you should hunt for the characters "MAG1". Enter the following commands to the debugger:

*** HL 200000 300000**

Pattern: 'MAG1' { use CAPITAL letters only }

When the debugger finds the first place in memory that this name exists it will display the line and address containing this name. Hit RETURN to have it search for the next one. Continue until the debugger returns with the message:

Not found

The name may actually exist in several places in mass memory, although only one of these locations will help you find the function code. If the name exists more than once, you must decide which is the “right one.” The address given at the beginning of the line must end with an “E.” This may help to eliminate some locations.

In the example the terminal looked like this after searching through memory:

```
00225332  4D 41 47 31 00 00 00 00 00 00 00 00 00 00 00 00 MAG1.....
0022928E  4D 41 47 31 00 00 00 00 00 00 00 00 00 00 00 00 MAG1.....
Not found *
```

The second entry is the only possible correct choice, since its address ends with “E.”

When you think you have found the right one (or want to see if it is correct), subtract Hex 1A from the address and list that address for 2 lines. If this is the entry you are looking for, you will notice that the first 4 bytes of the first and second lines contain the same mass memory address which must end in the number 4. When you have found this block, add Hex 12 to the address in the first 4 bytes of the first (and second) line. This is the location in mass memory where your user-written function code starts.

In the example, 1A is subtracted from 22928E to get 229274, then the following is typed in to examine this memory location:

```
*L 229274 229294
00229274  00 22 97 84 00 22 8B E4 00 00 00 00 02 00 04  ."...".d.....
00229284  00 22 97 84 00 00 00 00 00 04 4D 41 47 31 00 00  .".....MAG1..
*
```

Hex 12 is then added to the value found in the first 4 bytes:

$$00229784 + 12 = 00229796$$

This is the mass memory location of the beginning of the user-written function code.

4. By adding the offset found in step 2 and the start address found in step 3, calculate the actual address of the code in question.

$$00229796 + 2EC = 00229A82$$

This is the address inside the code where you want to set the breakpoint. It is usually a good idea to check the contents of memory with what you expect them to be at this location. You can list the memory at the address you have calculated and compare the numbers with the numbers which represent the instructions given in your listing file. If they do not match, then you do not have the correct address.

```
*L 00229A82 0229A92
```

```
00229A82 20 6E FF FC 22 7C 00 00 00 00 B1 C9 67 00 02 7E n.i"i....1Ig..↑  
*
```

Compare the contents of memory location with what you expected to see (see listing excerpt in step 2 above) and see that you have found the correct address.

5. Set breakpoint. Up to seven breakpoints can be set at one time.

***BR 229A82 ***

6. Type "G" to return to PS 390 run-time code. Just before the code is executed at the breakpoint location, the PS 390 will enter Debug mode. The picture on the display will go away, and the debug prompt as well as register contents will appear on the ASCII terminal screen. The program counter is listed with the registers. It is at the address of your breakpoint.
7. Return to PS 390 run time by typing "G" at the terminal, or trace through subsequent instructions as desired.

NOTES

While the PS 390 is in debug mode, all interrupts are disabled. Thus, any data entering the PS 390 from the host or peripheral devices will be lost. Therefore, make sure no data are coming in from the host at the time your breakpoint is executed. If your function relies on data from the host, one way to get around this problem is to:

1. Place an F:SYNC(2) function in front of your function.

2. Connect the input source to input <1> of the sync function and output <1> of the sync function to your user-written function.
3. Set input <2> of F:SYNC to be a constant.
4. When all data from the host have arrived on input <1> of the sync function, send a message to input <2>. This will cause the data to pass through the sync function and cause your user-written function to be executed.

Section AP8

User-Written Function Reference

8.1 Introduction

This part of the manual is provided as a reference section for information you may need while writing your own functions. It contains the following sections:

- **Message Types** Description of the legal message types that can be passed between functions. Also contains an excerpt from USERSTRUC.PAS that illustrates how the types are declared.
- **Utility Routines** Topical listing and short description followed by the utility routines in alphabetical order. Advanced User-Written Function procedures are described separately after the User-Written Function utilities.
- **Stack Usage** List of the stack usage, in bytes, of the utility routines.
- **Error Messages** A list and description of system error messages you might encounter while writing your own functions.

These routines are all declared in the E&S-provided file, USERSTRUC.PAS. The procedures themselves are in USERLINK.RO and must be linked to any function you write.

NOTE

During the initial distribution of USERSTRUC.PAS, two versions were sent to customer sites. The difference in the two versions is in the names of the pointers and the PS 390 floating-point record definition. This release of USERSTRUC.PAS supports the following naming conventions, with the strong recommendation that the conventions in this manual be used:

Preferred	Acceptable
Double	PS 390 floating point
UWFInQarray	InUWFAQarray

The two names shown below are no longer acceptable and must be modified:

Not Acceptable	Must Be Modified To
PtrInQarray	PtrUWFInQarray
InQarray	UWFInQarray

8.2 Message Types

The type declarations included in USERSTRUC.PAS define the various message types that are used in the PS 390, as well as other types that are used by the utility routines. This section describes these types and how to use them.

8.2.1 QDtype and Qdata

The QDtype is used to specify the different types of Qdata message blocks available in the PS 390 runtime system. Qdata blocks are the primary vehicle for communication between functions in the PS 390.

The Qdata record declaration specifies the formats for all messages. The first field, **Next**, is a pointer to the next message in a list or queue of messages. Ordinarily, you should not use the Next field explicitly. Next is defined as `Ptrqdata = ↑Qdata; {pointer to a message}`. Nearly all communication that takes place within the PS 390 runtime system occurs by passing message blocks (often referred to as Qdata blocks or a “Qdata”). The Ptrqdata contains the pointer to a Qdata block.

The **Qtyp** field indicates the **QDtype** of the message body. The remaining fields vary, depending on the contents of the Qtyp field. A general listing of the QDtypes follows:

QDtype	= { types of Qdata (message) blocks }		
(
{ 0}	Qreset,	{ dataless: reset a function instance	}
{ 1}	Qprompt,	{ dataless: flush the CI pipeline	}
{ 2}	Qboolean,	{ normal carrier of Boolean values	}
{ 3}	Qinteger,	{ normal carrier of integer values	}
{ 4}	Qreal,	{ normal carrier of floating point values	}
{ 5}	Qstring,	{ original carrier of byte strings, not used	}
{ 6}	Qpacket,	{ carrier of byte strings	}
{ 7}	Qmorepacket,	{ continuation Qpacket carrier of byte string	}

```

{ 8}      Qmove2,      { 2D vector including P bit          }
{ 9}      Qdraw2,      { 2D vector including L bit          }
{10}      Qvec2,        { 2D vector with no P/L bit (normal vector) }
{11}      Qmove3,      { 3D vector including P bit          }
{12}      Qdraw3,      { 3D vector including L bit          }
{13}      Qvec3,        { 3D vector with no P/L bit (normal vector) }
{14}      Qmove4,      { 4D vector including P bit          }
{15}      Qdraw4,      { 4D vector including L bit          }
{16}      Qvec4,        { 4D vector with no P/L bit (normal vector) }
{17}      Qmat2,        { 2x2 matrix                        }
{18}      Qmat3,        { 3x3 matrix                        }
{19}      Qmat4,        { 4x4 matrix                        }
{20}      Qusertype     { type that user may use to define own message }
);

```

QDtype is padded with 260 miscellaneous elements to ensure that a 16-bit field is allocated by the Pascal compiler rather than the 8-bit field that would be allocated otherwise.

Most of the QDtypes are self-explanatory. A few that may need more explanation follow.

Qreset and **Qprompt** are not ordinarily used by user-written functions. **Qstring** is obsolete and remains in the PS 390 system for historical purposes only. You should use **Qpacket** for string messages.

Qpacket is the standard byte/character message block. Although the Pascal declaration for a **Qpacket** indicates that the **P_Cnt** field will hold 255 characters, the actual amount of storage allocated varies. You should never attempt to reference characters outside of the **P_Cnt[P_Beg..P_Lth]**; doing so could cause a fatal error in the PS 390. If you need to append to the end of a **Qpacket**, you must allocate a new message large enough to hold the entire string.

Qmorepacket is used as a continuation block for a message coming from the host with more than 255 bytes of information. Again, this message type is not used by ordinary functions.

QuserType is available to allow users to define their own messages types, while still handling the messages uniformly within the PS 390 system. If you want **QuserType** messages to carry data, you must modify the declaration of the **Qdata** record type in **USERSTRUC.PAS** to include a variant for **QuserType** that contains the desired fields. **QuserType**'s **Qtyp** fields must be explicitly filled in by the program. (Refer to **F:SPIRO** in section 6 of this manual for an example of how to use **QuserType** messages.)

The QData record declaration (from USERSTRUC.PAS) follows.

```
Int16 = -32768..32767; { 16-bit integer }
Qdata =
  RECORD
    Next: Ptrqdata ; { next message in a list of messages }
    CASE Qtyp: Qdtype OF    { type of message }
      { Qreset: no datum carried }
      { Qprompt: no datum carried }
      Qboolean:
        (
          b: boolean
        ) ;
      Qinteger:
        (
          i: integer
        ) ;
      Qreal:
        (
          r: PS_390_floating_point
        ) ;
      Qpacket, Qmorepacket: { byte-string }
        (
          P_lth: int 16 ; { max byte number }
          P_beg: int 16 ; { min byte number }
          P_cnt: Bytespell { byte of message }
        ) ;
      Qmove2, Qdraw2, Qvec2
      Qmove3, Qdraw3, Qvec3
      Qmove4, Qdraw4, Qvec4:
        (
          V4: Vector    { all vectors use 4D indexing }
        ) ;
      Qmat2, Qmat3, Qmat4:
        (
          Mat4: Matrix { all matrices use 4x4 indexing }
        ) ;
    END ; { Qdata }
```

8.2.2 Input Message Pointers

```
PtrUWFInQarray = ↑UWFInQarray;
UWFInQarray = ARRAY [1..MaxInputQueues] of PtrQdata;
```

UWFInQarray contains the pointer to the input queues for the userwritten function. PtrUWFInQarray contains the pointer to the UWFInQarray.

8.2.3 PS 390 Floating-Point Numbers

In the PS 390, floating-point numbers are defined as

```
Double          : RECORD
c               : Int16 ; { Exponent }
m               : Integer ; { Fraction }
not_used        : Int16 { not used }
                END;
```

where:

c is the excess-1024 power of two of the number.

m is the mantissa in M68000 internal type “long” interpreted as a two’s-complement number whose binary point is between bit 31 (the sign bit) and bit 30.

The fraction is normalized; i.e., except for zero, the sign bit differs from the most significant bit. This normalization is accomplished by adjustment of the exponent for any shifts which might occur. Such adjustment can cause the exponent to exceed its maximum (overflow) or underrun its minimum (underflow).

Because the fraction is stored as a two’s-complement number, overflow can occur when negating the negative number of largest magnitude.

not_used is a field that is not used but must exist in all the floating-point records. This field is included in each floating-point record to ensure that data are aligned on 8-byte boundaries, a factor that can help improve some execution speeds.

Examples:

R : Double;

. . .

```
R.c := 0 + 1024;
R.m := 16384 * 65536;
{This is number 1/2 * 2**0 = .5}
```

```
R.c := -1+1024 ;
R.m := -2147483648
{This is the number -1 * 2**-1 = -.5}
```


Vector = ARRAY [0..3] of PS 390_Floating_Point;

All vectors (2D or 3D) are allocated as 4D vectors of floating-point values to allow X,Y,Z and I to be accommodated, if required. When a vector becomes part of a display data structure, it has been optimized appropriately.

Matrix = ARRAY [0..3, 0..3] of PS 390_Floating_Point;

All matrices (2x2, 3x3, 3x4, and 4x4) are allocated as a 4x4 matrix of floating-point values to allow all sizes of matrices to be accommodated, if required. When a matrix becomes part of a display data structure, it has been optimized appropriately.

Bytespell = ARRAY [1..255] of CHAR;

The Bytespell array is used to hold bytes/characters for Qpacket and Qmorepacket messages.

8.3 Topical Listing Of Utility Routines

Input Queue Handling and Function Scheduling Procedures

These procedures are provided to obtain access to input messages and control function scheduling. Functions are not required to look at all of the inputs, but there must be one message on each of the inputs for the function to run. The scheduling procedures are:

CkInputs
CleanInputs

Error Reporting Procedures

When a PS 390 function detects an error, a message is displayed. When this happens, the status of the messages on the input queues is the same as if the function had run to completion without an error. There is one exception: If the error was due to a message on a Cqueue, the message is removed.

The supplied utility procedures provide for the basic needs of error reporting. They are not intended to cover all cases. If the proper error message routine does not exist, it is the programmer's option to write a new error routine that meets that need.

The writer of the function is responsible for seeing that if one of these error routines is executed, flow immediately proceeds to the "IF CleanInputs

THEN” statement after all the inputs have been checked for errors. This follows the error philosophy that requires a function to have one complete input set to execute, and if anything is in error in that set, the function will not run.

The following error handling procedures are provided in USERLINK.

- QIIMessage
- QIIValue
- Qincompatmsgs
- Systemerror
- UWError

Set_Cness Procedure

There is one procedure provided to change the Cness or Tness of a function queue:

- Set_Cness

Private Data Queue Procedures

Some functions require that data acquired during the process of the function be retained from execution to execution. These functions are referred to as having “private data queues”; that is, queues for data not fed from input to output during each execution cycle. No outside function can send messages to these queues. Procedures provided for functions with private data queues are:

- CkPrivate
- SavePrivate

Message Management Procedures

The following procedures are used to send, copy, or dispose of messages:

- DropMessage
- MsgCopy
- QSendCopyMess
- SendMsg

PS 390 Floating-Point Utilities

The procedures to perform floating-point computations are:

- FCadd
- FCdivide
- FCint2double
- FCinteger
- FCmultiply
- FCnearzero
- FCp2multiply
- FCround
- FCsquareroot
- FCsubtract
- Fpabs
- Fpecomp
- Sincos

Carving and Initialization Procedures

The following procedures are used to carve and initialize new data types:

- Newqboolean
- Newqinteger
- Newqmatrix
- Newqnil
- Newqpacket
- Newqreal
- Newqvector
- Newtry

Timing Procedures

- Csecs
- Frames
- Hrtime
- Ticks

String Handling Procedures

- Char_text
- Int_text
- Real_text
- Text_text
- Time_text

Other Procedures Provided Via USERLINK.PAS

My_in_out
My_name
Rndmnumber
Vfetch
Vstore

Advanced User-written Function Procedures

Lk_cursuffix
Lk_nosuffix
Lgaupdate
Announceupdate
Msgstore
Setlock
Clrlock
Incausage
Decausage
AcpProof
Acpprf1
OLbaddtoreset
Removefromset
FetchBlock
Acp_v3f
Acp_v2f
Acp_v3b
Acp_v2b
nStoreVector
nNewAcpdata
Store3x3
Store4x4
Drop_name
GetVector
Rawbacopy
Rawbcopy
Rawchcopy
Size_of
FetchAdnum
nFetchCopy
WaitFrame

loc_thead
ptr_dcb
DropNE
Newreturns
Reactivate
Myanyoutputs
Pushmyinput
WaitCsec
HA_cursor
HA_no_cursor

8.4 Procedures Provided Via USERLINK

CkInputs

FUNCTION CkInputs (Nmin, Nmax : Int16) : PtrInQarray;

CkInputs sets a pointer to each of the input queues specified in the inclusive range Nmin to Nmax and stores them in an array. If there is a message on each of the input queues, it returns a pointer to the array and the function state is changed to MIDRUNNING. This signifies that the function may execute. The function returns NIL if there are queues in the range that do not have a message. When NIL is returned, the function is put into the MSG_WAIT state and must exit.

A function does not have control over these input message blocks. For example, it cannot reuse an input message block. Data being sent out must be contained in newly created message blocks. If a function is to send a message through without change (such as F:SYNC), the utility procedure Qsendcopymess should be used for efficiency.

Char_text

PROCEDURE Char_text (c: char; VAR b,e: Int16; VAR ca: Bytespell) ;
FORWARD ; { TEXTUTIL.PAS }

Char_text adds one character to a text string ca at location b+1 within that string but not beyond location e. b is updated. If b is negative, system error 81 is generated.

CkPrivate

FUNCTION CkPrivate : Ptrqdata;

Ckprivate returns a pointer to the private message for this function, if it exists, or returns NIL if the private message does not exist.

CleanInputs

FUNCTION CleanInputs : Boolean;

CleanInputs must be called after the input messages have been processed and the outputs have been sent. Its purpose is to “clean up” the input queues and determine whether the function may run again immediately.

This procedure can recognize whether an input queue is a trigger queue (Tqueue) or a constant queue (Cqueue). It drops the first message from each Tqueue and leaves Cqueues unchanged.

The function must be in the MID_RUNNING state when this utility procedure is called. If the function can run again immediately, CleanInputs returns TRUE and the function state is set to RUNNING. If there are not enough input messages for the functions to run again, FALSE is returned and the function state is set to MSG_WAIT. It is also possible for CleanInputs to return FALSE if the function has been running longer than 2 milliseconds; in this case, the state is set to ACTIVE and is required to give up control so that other functions can run.

Csecs

FUNCTION Csecs: Integer ;

Csecs returns the number of centiseconds since the system was booted.

DropMessage

PROCEDURE DropMessage (VAR m: Ptrqdata) ;

Dropmessage disposes of the message m. It should be used rather than DISPOSE for all message dropping--especially for dropping messages of unknown Qtype--since it knows when additional data items are affected by the message being dropped. The message m is no longer the property of the calling code. m is set to NIL.

A function does not need to dispose of input messages explicitly since the procedure CleanInputs disposes of them.

FCadd

PROCEDURE FCadd (VAR augend, addend: Double;
VAR sum: Double);

FCadd does a floating-point add.

FCdivide

PROCEDURE FCdivide (VAR dividend, divisor: Double;
VAR quotient: Double);

FCdivide does a floating-point divide. If the divisor is zero, the function returns the largest positive number if the dividend is positive, otherwise it returns the largest negative number.

FCint2double

PROCEDURE FCint2double (num : Integer; VAR floated: Double);

FCint2double makes a floating-point number from an integer.

FCinteger

PROCEDURE FCinteger (VAR innum: Double; VAR outnum: integer);

FCinteger truncates a floating-point number to an integer.

FCmultiply

PROCEDURE FCMultiply (VAR a, b: Double;
VAR product: Double) ;

FCmultiply does a floating-point multiply.

FCnearzero

FUNCTION FCnearzero (VAR tiny : Double; negpower2 : Int16 : Int8 ;)
{ negpower2=1 --> within .5; =2 --> within .25}

Returns a byte that indicates if a number is close to zero given an absolute tolerance.

The tolerance (negpower2) is expressed as the negative power of two; that is, 0 means that anything less than 1 is close enough,

1 means anything less than .5 is close enough,
2 means anything less than .25 is close enough, etc.,
and -1 means anything less than 2 is close enough, etc.

Results mean:

-1 means the number is not close to zero, and it is negative,
0 means the number is close enough,
1 means the number is not close to zero, and it is positive.

FCp2multiply

PROCEDURE FCp2multiply (VAR innum: Double; power: Integer;
VAR outnum: Double) ;

FCmultiply multiplies innum by 2 raised to the power specified by power, (i.e., power is added to the exponent of innum, hence power can be either positive or negative.)

FCround

PROCEDURE FCround (VAR Innum: Double; VAR outnum: Integer) ;

FCround rounds a floating-point number to an integer.

FCsqrt

PROCEDURE FCsqrt (VAR a: Double;VAR sqroot: Double) ; j FCsqrt returns the square root of a positive floating-point number. If the number is negative, 0 is returned.

FCsubtract

PROCEDURE FCsubtract (VAR minuend, subtrahend: Double;
VAR difference: Double) ;

FCsubtract does a floating-point subtract.

Fpabs

PROCEDURE Fpabs (VAR r: Double) ;

Fpabs changes r to the absolute value. This is a destructive operation in that it changes r itself and does not put the absolute value in another variable.

Fpecomp

FUNCTION Fpecomp (VAR x1,x2: Double): Int8 ;

Fpecomp compares two floating-point numbers and returns:

-1 if $x1 < x2$
0 if $x1 = x2$
1 if $x1 > x2$

Frames

FUNCTION Frames: Integer ;

Frames returns the number of frames displayed since the system was booted.

HRTIME

PROCEDURE HRTIME (VAR c,f,d: Integer) ;

The HRTIME procedure returns a high-resolution clock value. It returns the current time in centiseconds, and a fraction indicating the amount of time remaining until the next centisecond.

It is mainly used to calculate the elapsed time between two events, as shown in the following example:

```
HrTime (c0, f0, d);           { initial high-resolution time }
•
•
•
HrTime (c1, f1, d);           { final high-resolution time }
Elapsedtime := ((c1-c0)*d) + (f0-f1); { actual runtime of code }
```

c is the current value of the centisecond clock, returned as a 32-bit integer which wraps around to zero. f/d is the fraction remaining until the next centisecond. Note that f decreases in value for increasing time, while c increases in value for increasing time.

Int_text

PROCEDURE Int_text (n: Integer; Ns,Nz: Int16;
VAR b,e: Int16; VAR Ca: Bytespell) ;

Int_text converts an integer to text, as a signed decimal number, and adds it to a text array, via Char_text. Ns is the minimum number of characters to generate, and Nz is the minimum number of leading zeros to print. (Note: to print n=0, Nz must be 1.) The number starts in Ca[b+1] and will not go beyond Ca[e]. b will be changed. e specifies last character which can be changed. If b is negative, system error 81 is generated.

MsgCopy

FUNCTION Msgcopy (m: Ptrqdata): Ptrqdata ;

Msgcopy makes a copy of the message m. The message returned by Msgcopy is the property of the calling code and must be disposed of (Dropmessage) or handed on (Sendmsg) before the calling code returns.

My_in_out

PROCEDURE My_in_out (VAR N_in,N_out: Int16) ;

My_in_out reports the number of input queues and output ports for the current function instance.

My_name

FUNCTION My_name : Ptrqdata ;

My_name looks up the name of the function instance and returns that name in a Qpacket.

Newqboolean

FUNCTION Newqboolean: Ptrqdata ;

Newqboolean carves and initializes a Qboolean message. Initialization includes setting: Qtyp = Qboolean, value to FALSE.

Newqinteger

FUNCTION Newqinteger: Ptrqdata ;

Newqinteger carves and initializes a Qinteger message. Initialization includes setting: Qtype = Qinteger, value to zero.

Newqmatrix

PROCEDURE Newqmatrix(Typ: Qdtype): Ptrqdata ; { Qmat2, ... }

Newqmatrix carves and initializes a matrix message of type Typ and contents all zero (Note: not floating-point number zero).

Newqnil

PROCEDURE Newqnil(Typ: Qdtype): Ptrqdata ; { Qreset; Qprompt }

Newqnil carves and initializes a dataless message of type Typ. Initialization includes setting: Qtype = Typ

Newqpacket

PROCEDURE Newqpacket (Typ: Qdtype; { Qpacket or Qmorepacket }
Nbytes: Int16): Ptrqdata ;

Newqpacket carves and initializes a Qpacket or Qmorepacket message large enough to hold Nbytes of information. Note that, although the Pascal declaration for a Qpacket indicates that the P_Cnt field will hold 255 characters, only Nbytes bytes are actually allocated by NewQPacket. You should never attempt to reference characters beyond the end of the string. Initialization includes setting: Qtype = Typ, P_lth = Nbytes, P_beg = 1, P_cnt[1..Nbytes] = 0

Newqreal

FUNCTION Newqreal: Ptrqdata ;

Newqreal carves and initializes a Qreal message. Initialization includes setting: Qtype = Qreal, exponent and mantissa to zero (Note: not floating-point value zero).

Newqvector

FUNCTION Newqvector (Typ: Qdtype): Ptrqdata ; { Qvec2, ... }

Newqvector carves and initializes a vector message of type Typ and contents all zero (Note: not floating-point number zero).

Newtry

FUNCTION Newtry (num_bytes : Integer) : Ptrqdata;

Newtry returns a block of the specified length in bytes if one is currently available in the system. If one is not available, NIL is returned. This differs from all the other “Newq” functions that will not return until a block of the specified type is available (or eventually cause the system to crash with a trap 0). Newtry is used to carve data blocks when the function has a choice of what to do if a data block of sufficient size is not available.

QillMessage

PROCEDURE QillMessage (Inqueue : Int16);

The parameter Inqueue indicates the input queue that contains the bad message. QillMessage prints the message:

Message which function cannot handle.

It then drops the message and sets INPUTS↑.[inqueue] := NIL;

QillValue

PROCEDURE QillValue (Inqueue : Int16);

The parameter Inqueue indicates the input queue that contains the bad message. QillValue prints the message:

Type okay but value out-of-range

It then drops the message and sets INPUTS↑.[inqueue] := NIL;

Qincompatmsg

PROCEDURE Qincompatmsg (one : Int16; theother : Int16);

The parameters indicate the input queues that are incompatible. This procedure prints the message:

Incompatible message types detected by this function

It then deletes the message on the queue theother and sets INPUTS↑.[theother] := NIL;

QSendCopyMess

PROCEDURE QSendCopyMess (inqueue, outqueue: Int16);

QSendCopyMess takes the message on the specified Inqueue and sends it unchanged on the specified Outqueue of the function. If the queue is a Cqueue, a copy of the message is sent. If it's a Tqueue, the message is removed and sent.

NOTE

Consuming a message results in INPUTS↑.[Inqueue] := NIL and any following references to INPUTS↑.[Inqueue] will produce unpredictable results. Because of this, it is recommended that QSendCopyMess only be used prior to the "IF CleanInputs THEN" statement.

Real_text

PROCEDURE Real_text (VAR r: Double;VAR b,e: Int16;
VAR Ca:Bytespell);

Real_text converts r into text, expressing roughly 5 digits. If possible, r is printed as a fixed point number, but if it is too small or too large, it is printed in exponential form. The real number will be written starting at Ca[b+1]. b will be changed. e specifies the last character that can be changed. If b is negative, system error 81 is generated.

Rndmnumber

FUNCTION Rndmnumber (seed : Integer): Int16;

Returns a pseudo-random 16-bit number. If the value of the seed is zero, then the value returned is computed based on the current seed value. If the passed seed value is non-zero, then it is made into the current seed and then the number is computed. The linear feedback shift register technique uses a 31-bit seed (bits 31 to 1) with taps on bits 31 and 6. This algorithm does 7 bits at a time (3 times for 16 bits). It does not repeat until $2^{31} - 1$ iterations.

SavePrivate

PROCEDURE SavePrivate (msg : Ptrqdata);

This utility procedure is used for filling the Private queue, a queue to which no outside function can send messages. Once a message has been saved on the private queue, it will remain there as long as the function exists. The value, however, can be changed, and the queue will then retain the new value.

SendMsg

PROCEDURE SendMsg (VAR m: Ptrqdata; o: Int16) ; Sendmsg sends the VAR m to any and all other named entities connected to the function's output port <o>. After this is done, the message m is no longer the property of the calling code and m has value NIL.

Set_Cness

PROCEDURE Set_Cness (input : Int16; cqtype: Boolean); Set_Cness allows the function itself to do the SETUP CNESS TRUE/FALSE<i>a; command. input is the input number of the function that the Boolean value will be sent to. ctype is the Boolean value: TRUE sets the specified input to a Cqueue and FALSE sets the specified input to a Tqueue. By default, when a function is instanced, all input queues are Tqueues.

WARNING

Using this procedure inside the function, as well as sending the SETUP CNESS command to the function, may produce unpredictable results because it may not be clear which code is executed last. Either one method or the other should be used, not both.

Sincos

PROCEDURE Sincos (Angle: integer; VAR sine: Double;
VAR cosine: Double;

Sincos computes the sine and cosine of an angle. angle is an integer between 0 and 65535, corresponding to the range 0 to 2*pi radians. It computes the sine of angle by using the most significant 8 bits to index into a table of values and using linear interpolation of the least significant 8 bits. If angle is not in the first quadrant, it is converted to an angle in the first quadrant using trigonometric relations.

RESULT := TABLE(A) + B * 256 * (TABLE(A + 1) - TABLE(A))

A = most significant 8 bits of angle

B = least significant 8 bits of angle

The cosine is computed by adding 90 degrees to the angle, then computing the sine.

Systemerror

PROCEDURE Systemerror (n: Int16) ;

Systemerror crashes the PS 390 with a TRAP 6. The parameter n becomes the system error number. There is no return from a Systemerror call.

Text_text

PROCEDURE Text_text (VAR B1,E1: Int16; VAR Ca1: Bytespell;
VAR B2,E2: Int16; VAR Ca2: Bytespell) ;

Text_text will copy characters from Ca1 to Ca2 starting from B1+1 in Ca1 into B2+1 in Ca2 and continuing until either Ca1[E1] has been copied or Ca2[E2] has been changed, at which point copying stops. Both B1 and B2 are changed. If B1 is negative, System error 80 is generated.

Ticks

FUNCTION Ticks: integer ;

Ticks returns the number of ticks (line-clock ticks 120Hz or 100 Hz) since the system was booted.

Time_text

PROCEDURE Time_text (n: integer; VAR b,e: Int16; VAR Ca: Bytespell) ;

Time_text converts n number of seconds to text as a time. In the form: dd hh:mm:ss. The time string will be written starting at Ca[b+1]. b will be changed. e specifies the last character which can be changed. If b is negative, system error 81 is generated.

UWError

PROCEDURE UWError (VAR msg : Ptrqdata);

UWError allows the user-written function to display any type of error message desired. msg must be a Qpacket containing the characters of the message to be printed. msg is set to NIL by the procedure.

Vfetch

FUNCTION Vfetch (name: Ptrqdata) : Ptrqdata; { a Qpacket }

Vfetch fetches the contents of a named variable. The name of the VAR is supplied in name (Qpacket). If any error is detected, Vfetch returns NIL. Otherwise, Vfetch returns a copy of the message stored in the specified VAR. The returned message is owned by the function.

Vstore

PROCEDURE Vstore (name: Ptrqdata; VAR new_val: Ptrqdata) ;

Vstore stores the Qdata new_val into a named variable. The name of the variable is supplied in name (must be a Qpacket). If the store succeeds, new_val is set to NIL and is no longer owned by the function. Otherwise it is left alone.

8.5 Advanced UWF Procedures

Lk_cursuffix

```
FUNCTION Lk_cursuffix(  
    Nlth: integer; { name length }  
    VAR    Nspell: Namespell { name to be looked up }  
): Ptralphablk;
```

Lk_cursuffix returns a pointer to the block of memory that contains the name of an object and pointers to its definition. The suffix of the currently running function is appended to the end of the characters in Nspell. If the name currently exists a pointer to it is returned. If it does not exist a new name block is created and the pointer to that is returned. This routine can then introduce new names into the system. Since names may be multiply referenced, a usage count is maintained. Only when that usage count goes to zero is the name block disposed. IT IS VERY IMPORTANT TO KEEP THIS COUNT ACCURATE! If the count is too small it will crash the system at some later time, if it too large the memory will never be recovered. Since this routine returns another reference to the name, the usage count has been incremented for that value. If the reference is not stored, when finished the usage count should be decremented. Changing the usage count is done through the routines Incausage and Decausage. Nlth is the number of characters in the name and Nspell is the array of character for the name.

NOTE

All names are uppercase, so any lowercase letters in Nspell will be changed to uppercase.

Lk_nosuffix

```
FUNCTION Lk_nosuffix(  
    length: Integer;    { name length NOT counting suffix }  
    Cinum: Int8;        { creating CI }  
    suffix: Char;       { the user suffix }  
    VAR    Nspell:Namespell { name to be looked up }  
    ): Ptralphablk;
```

Lk_nosuffix performs the same task as Lk_cursuffix except it is more general. Cinum is the number given to the name if it is created. This is used for such things as the INITIALIZE command. A CI may only initialize those things which it has created. Suffix is the suffix character added to the name. '0': user 1 hidden names, '1': user 1 accessible names, '2': user 2 hidden names, '3': user 2 accessible names. Nspell contains the character string of the name.

Lgaupdate

```
PROCEDURE Lgaupdate(  
    name: Ptralphablk;    { alpha of object to change }  
    data: Ptrnamedentity; { new data to be referenced }  
    VAR    Uph,Upt: Ptravupblk { head and tail of list of updates}  
    );
```

Lgaupdate includes an update request onto a list of updates which are to be performed. Updates are identification for the display processor that an object is to change. This procedure creates the proper information so that the name of the element pointed to by 'name' will have its data changed to now point to 'data'. The value for 'name' is obtained through the use of the Lk_cursuffix routine.

This routine can be called several times followed by a call to Announceupdate.

NOTE

Uph and Upt must be initialized to NIL prior to the first call of this routine.

Announceupdate

```
PROCEDURE Announceupdate(  
    VAR Uph,Upt: Ptravupblk    { head and tail of list of updates }  
);
```

Announceupdate takes the list of updates generated through the use of the Lgaupdate function and passes them to the display processor such that on the next frame the new definitions will be displayed and the old ones deleted.

Msgstore

```
PROCEDURE Msgstore(  
    Msg: Ptrqdata;    { pointer to message block }  
    a: Ptralphablk;    { alpha to receive message }  
    n: integer        { input to receive message }  
);
```

Msgstore is used instead of Sendmsg when the message is to be sent to a known destination and input and not to the list on one of the functions outputs. Msg is the message to be sent.

NOTE

Even though it is not a VAR parameter, msg is used just like in Sendmsg so the caller should treat it just as though it was set to NIL by Msgstore. a points to the name block for the recipient and n is the input number.

Setlock

```
PROCEDURE Setlock(  
    VAR x: Lock    { lock to be changed }  
);
```

Setlock sets a lock to be True.

Clrlock

```
PROCEDURE      Clrlock(  
    VAR      x: Lock      { lock to be changed }  
    );
```

Clrlock sets a lock to be False.

Incausage

```
PROCEDURE Incausage(  
    a: Ptralphblk { alpha to be incremented }  
    );
```

Incausage increases the usage count of a name block by one.

Decausage

```
PROCEDURE Decausage(  
    a: Ptralphblk { alpha to be decremented }  
    );
```

Decausage decreases the usage count of a name block by one. If the usage count becomes 0 the name block and data pointed to by the name block is disposed.

AcpProof

```
PROCEDURE AcpProof(  
    VAR location: ptracpblk; { pointer to be replaced }  
    newval: ptracpblk      { new pointer }  
    );
```

Acpprf1

```
PROCEDURE Acpprf1(  
    VAR    location: ptrsavstate;    {pointer to be replaced }  
          value: ptrsavstate        { new pointer }  
    );
```

AcpProof, Acpprf1 allow the changing of pointers directly without going through the normal update mechanism. Since pointers are 32 bits and the 68000 only writes 16 bits at a time, this procedure writes the new pointer in such a way that if the display processor is reading at the same time it is being written, the worst that could happen is that it would appear as a NIL pointer and the display processor will terminate traversal at that point and return to another branch it has to traverse. This means you may loose part of your picture for one frame. If the UWF does not use this routine and writes the new value directly, the display processor may be sent to a random place in memory and almost guarantee that memory will be corrupted.

OLbaddtoreset

```
PROCEDURE OLbaddtoreset(  
    A_son: Ptralphablk;    { alpha of branch to be added }  
    A_father: Ptralphablk; { alpha of set node }  
    VAR    Uph, Upt: Ptravupblk; a head and tail of list of updates}  
    VAR    Error: boolean;      { status on return }  
          Optimize : boolean    { optimize structure in effect }  
    .    );
```

OLbaddtoreset adds another branch in the traversal tree. Setnodes are the points in the traversal where the display processor branches and also the point at which saving the state and restoring it are performed. The recommended way of initially creating one of these branches is to Lgaupdate A_father to point to a Setnode which has had all of its pointers set to NIL. Olabaddtoreset is called to add the first and all subsequent branches at this point. A_son points to the beginning of the new branch. A_father points to the name of the Setnode (branch). Uph and Upt are the list of updates generated, just as Upt and Uph of Lgaupdate. Hence for the change to actually appear Announceupdates must be called. Error is returned TRUE if Father is not a set node. Optimize should be TRUE if the operation should be treated as though OPTIMIZE STRUCTURE; was in effect, otherwise it should be FALSE.

Removefromset

```
PROCEDURE Removefromset(  
    A_father: Ptralphabl; { alpha of set node }  
    A_son: Ptralphabl;    { alpha of branch to remove }  
    VAR Uph, Upt: Ptravupblk; { head and tail of list of updates }  
    VAR Error: boolean      { status on return }  
);
```

Removefromset removes the branch of A_father which points to A_son. Uph and Upt are the list of updates generated, just as Upt and Uph of Lgaupdate. Hence, for the change to actually appear Announceupdates must be called. Error is returned TRUE if Father is not a set node.

FetchBlock

```
FUNCTION FetchBlock(  
    block: PtrNamedentity; { pointer to block for which }  
                                { updated values are desired }  
    upt: Ptravupblk        { tail of list of updates }  
): Ptrnamedentity;
```

Fetchblock searches the set of updates pending and returns a copy of block which has all updates applied to it. block is the pointer to the block for which updated values are desired. upt should be NIL unless some updates have been generated which have not yet been 'announced'.

Acp_v3f

```
PROCEDURE Acp_v3f(  
    VAR v: Vector;      { vector to be converted }  
    VAR acpv: Vec3f;    { APC format result }  
    pl: boolean         { position/line }  
);
```

Acp_v3f converts a GCP format vector v to Vec3f0 format which is the format for 3D vector-normalized vectors. IF pl is TRUE the vector is a line, if FALSE the vector is a position.

Acp_v2f

```
PROCEDURE Acp_v2f(  
    VAR      v: Vector;      { vector to be converted }  
    VAR      acpv: Vec2f;    { APC format result }  
    pl: boolean { position/line }  
);
```

Acp_v2f converts a GCP format vector *v* to Vec2f0 format which is the format for 2D vector-normalized vectors. IF *pl* is TRUE the vector is a line, if FALSE the vector is a position.

Acp_v3b

```
PROCEDURE Acp_v3b(  
    VAR      v: Vector;      { vector to be converted }  
    VAR      acpv: Vec3b;    { APC format result }  
    pl: boolean; { position/line }  
    exp: Int16 { exponent to use }  
);
```

Acp_v3b converts a GCP format vector *v* to Vec3b0 format which is the format for 3D block-normalized vectors. IF *pl* is TRUE the vector is a line, if FALSE the vector is a position. *exp* is the exponent for the entire block. If *exp* is too small for a vector a fatal error will occur.

Acp_v2b

```
PROCEDURE Acp_v2b(  
    VAR      v: Vector;      { vector to be converted }  
    VAR      acpv: Vec2b;    { APC format result }  
    pl: boolean; { position/line }  
    exp: Int16 { exponent to use }  
);
```

Acp_v2b converts a GCP format vector *v* to Vec2b0 format which is the format for 2D block-normalized vectors. IF *pl* is TRUE the vector is a line, if FALSE the vector is a position. *exp* is the exponent for the entire block. If *exp* is too small for a vector a fatal error will occur.

nStoreVector

```
PROCEDURE nStoreVector(  
    VAR   block: Ptrnamedentity;    { block to store vector in }  
    VAR   v: Vector;                { the vector }  
        pl: boolean;                { position or line? }  
        firstblock: Ptrnamedentity  { first block in list for }  
);                                  { block-normalized }
```

nStoreVector is the recommended way to create vector lists. Initially create the first block with nNewAcpdata and then use nStoreVector to put all vectors into the data block. Block is the current block to which items are added. Initially block = firstblock but may change as vectors are added. v is the vector to be added. pl is TRUE if the vector is a line and FALSE if it is a position. firstblock is the pointer returned by nNewAcpdata and is the pointer to be used for Lgaupdate. DO NOT PASS the same variable for both block and firstblock as this procedure may generate a linked list for this particular vector list.

nNewAcpdata

```
FUNCTION nNewAcpdata(  
    n: int16;          { number of elements }  
    t: Dattype         { data type of block }  
): Ptrnamedentity ;
```

nNewAcpdata carves the requested data type to the appropriate size. n is the number of elements, vectors or characters. t is the type of namedentity, vec3f0, vec2f0, dstring, etc.

Store3x3

```
PROCEDURE Store3x3(  
    VAR   m: Matrix3;          { GCP format matrix }  
    Ob: Ptrnamedentity; { pointer to matrix operate node }  
    n: int16                   { offset into operate node to begin store }  
);
```

Store3x3 converts the GCP format matrix and stores it into a Matcon3 operate node which has been previously created starting at the location specified by n.

Store4x4

```
PROCEDURE Store4x4(  
    VAR m: Matrix4;           { GCP format matrix }  
    Ob: Ptrnamedentity; { pointer to matrix operate node }  
    n: int16                  { offset into operate node to begin store }  
);
```

Store4x4 converts the GCP format matrix and stores it into a Matload4 operate node which has been previously created starting at the location specified by n.

Drop_name

```
PROCEDURE Drop_name(  
    a: Ptralphabl { name to remove from dictionary }  
);
```

Drop_name removes the name block a from the dictionary such that Lk_cursuffix and Lk_nosuffix will not be able to located it.

GetVector

```
PROCEDURE GetVector(  
    block: Ptrnamedentity; { from which block }  
    index: Int16;           { which vector }  
    VAR v: Vector;          { returns the vector }  
    VAR pl: Boolean         { is it a draw? }  
);
```

GetVector converts a display processor format vector into a GCP format vector. block is the block which contains the vector, index is the number of the vector within that block, v is the GCP format returned vector and pl is returned TRUE if it is a line and FALSE if it is a position.

Rawbacopy

```
PROCEDURE Rawbacopy(  
    Nbytes: int16;      { number of bytes to copy }  
    VAR Inba,Outba: int8 { input and output buffers }  
);
```

Rawbcopy

```
PROCEDURE Rawbcopy(  
    Nbytes: int16;      { number of bytes to copy }  
    VAR Inba:char;      { input buffer }  
    VAR Outba: int8     { output buffer }  
);
```

Rawhcopy

```
PROCEDURE Rawhcopy(  
    Nbytes: int16;      { number of bytes to copy }  
    VAR Inba,Outba: char { input and output buffers }  
);
```

The above procedures allow for the copying of bytes from one block to another. PASCAL can be tricked by selecting the appropriate data types for Inba and Outba according to the particular need.

Size_of

```
FUNCTION Size_of(  
    b: Ptrnamedentity { pointer to block to check }  
): integer;
```

Size_of returns the number of bytes available in the particular chunk of memory. b can be any type of element: Ptralphanblk, Ptrnamedentity, etc. It must be declared properly for PASCAL to accept it.

FetchAdnum

```
FUNCTION FetchAdnum(  
    f: Ptrnamedentity; { pointer to Named Entity for whose }  
                        { .Adnum field search is being made }  
    apt: Ptravupblk    { tail of list of updates }  
): Int16;
```

FetchAdnum searches and returns the current .Adnum (count) field of the pointer namedentity which will be its value once all updates have been performed. f is the Ptrnamedentity for whose .Adnum field the search is being made. apt should be NIL unless there are updates which have not been 'announced'.

nFetchCopy

```
FUNCTION nFetchCopy(  
    block: PtrNamedentity; { vec list to copy from }  
    start: Int16;           { vector to start at }  
    count: Int16;          { how many vectors to copy }  
    upt: Ptravupblk        { update list tail }  
): PtrNamedentity;
```

nFetchCopy searches the update list to return a current copy of the particular block of a vector list as it will be when all updates are completed. block is the block as currently in memory. start is the index of the vector to start at, count is the number of vector to return. upt should be NIL unless the function has updates which have not been 'announced'. The function returns a block which contains the vector lists requested.

WaitFrame

```
PROCEDURE WaitFrame(  
    framecnt : integer { number of frames to wait }  
);
```

WaitFrame puts the function in I/O wait and waits for the number of frames specified in framecnt before reactivating the function.

loc_thead

FUNCTION loc_thead : Ptrcommhead;

Loc_thead returns a pointer into mass memory of the location of the Comthead.

ptr_dcb

FUNCTION ptr_dcb : Ptrdcb;

Ptr_dcb returns a pointer into mass memory to the current users DCB.

DropNE

PROCEDURE DropNE(
 f : Ptrnamedentity { name to be disposed of }
);

DropNE disposes of the Named Entity f and any associated blocks.

Newreturns

PROCEDURE Newreturns(
 givemenil : boolean { return NIL when not enough memory }
);

Newreturns sets a flag that tells the free storage routines how to handle a request for memory if there isn't enough to grant the request. If givemenil is TRUE and there is not sufficient memory a NIL pointer is returned on the request. If givemenil is FALSE then the free storage routines will wait for a block to become available (eventually crashing the system if not found). Any NEW that needs to have this flag set TRUE should be followed immediately with a call to this routine to set it FALSE for other functions.

Reactivate

PROCEDURE Reactivate ;

Reactivate puts the current function on the active list. The function must exit immediately after this procedure call.

Myanyoutputs

FUNCTION Myanyoutputs(
 n: int8 { output number to check }
): Boolean;

Myanyoutputs checks the output specified in n to see if there are any connections to that output. If there are no connections a FALSE is returned. This can be used to determine the usefulness of doing some large calculation for an output that has nothing connected to it.

Pushmyinput

PROCEDURE Pushmyinput(
 m: Ptrqdata; { data to be pushed on input }
 n: Int8 { input number to push data on }
);

Pushmyinput causes the data pointed to by m to become the first message on the input queue n.

WaitCsec

PROCEDURE WaitCsec(
 csecnt : integer { number of centiseconds to wait }
);

WaitCsec puts the function in I/O wait and waits for the number of centiseconds specified in csecnt before reactivating the function.

NOTE

The following two routines should only be called if the system is a PS 390. This can be determined by looking at the field of the DCR called DCR^.system. This is of type *systemtype* which has the values sysPS300, sysPS350, sysPS390 as defined in GLOTYPES.SA. If either of these routines is called on a 330 or 350 system, the system will crash with a bus error.

HA_cursor

```
PROCEDURE HA_cursor(  
    VAR x,y: Double; VW390 : Ptralphablk  
);
```

This routine will position the currently selected hardware cursor at the position specified within the selected viewport.

x and y are the screen space x and y coordinates for the cursor. **VW390** is a pointer to the viewport operation node within which the cursor is to be displayed. If a NIL pointer is passed, a default viewport of 864 x 864 pixels is used.

HA_no_cursor

```
PROCEDURE HA_no_cursor;
```

This routine turns off the hardware cursor.

8.6 Stack Size

This section contains the rough stack usage estimates of some of the utility routines. These estimates are used to come up with an allocation scheme to provide enough stack space for the execution of user-written functions.

The UWF stack is allocated when functions are downloaded to the PS 390. As each function is processed by the SREC_GATHER function, the stack size requested is checked against the size of the currently allocated stack. If the requested stack size is the same or smaller, no action is taken. If the requested stack size is larger than the currently allocated stack, the current stack is disposed and a new one is allocated. All UWFs therefore use the same stack area.

It is very important that adequate stack size be allocated for any function you write. Failure to do so will cause the PS 390 to crash when the function is activated. It is better to overestimate the stack requirements than to underestimate and risk a crash.

It is a good idea to load the UWF with the largest stack first. A good initial estimate is 500 to 1000 bytes. This allows for the overhead of the utility routines and for an ordinary number of local variables. If your function has many local variables, especially matrices or large arrays, you should add the memory requirements for these variables. If you have included local procedures that may be called recursively, you should multiply the amount of stack usage for each procedure by the maximum depth of recursion and add that amount to the total.

A stack size of 1000 bytes is adequate for all of the function examples included in this manual with the exception of F:BEZIER, which requires about 5000 bytes.

Procedure	Stack Use In Bytes
CSecs	22
Char_text	22
CkInputs	36
CleanInputs	42
DropMessage	22
FCadd	22

(Rough stack usage estimates continued.)

Procedure	Stack Use In Bytes	
FCdivide	22	
FCinteger	22	
FCint2double	22	
FCmultiply	22	
FCnearzero	22	
FCp2multiply	18	
FCround	8	
FCsqrt	22	
FCsubtract	18	
Fpabs	22	
Frames	22	
HRtime	18	
Int_text	52	(1 character)
Int_text	94	(2 characters)
Int_text	290	(10 characters)
MsgCopy	40	
My_in_out	18	
My_name	66	
Newqboolean	24	
Newqinteger	24	
Newqmatrix	26	
Newqnil	26	
Newqpacket	28	
Newqreal	24	
Newqvector	32	
Newtry		
QSendCopyMess	240	
QIILMessage	140	
QIILValue	140	
Qincompatmsgs	154	
Real_text	310	(E-format)
Real_text	293	(F-format)
Rndmnumber	22	
SendMsg	152	

(Rough stack usage estimates continued.)

Set_Cness	180
Sincos	22
Text_text	42
Ticks	22
Time_text	134
UWError	22
Vstore	124

8.7 Error Messages

The following list gives the error message, a brief description, and a short summary of what might have caused the error. It is provided as a simple guideline to some of the more common mistakes made in writing functions. A complete list of PS 390 errors is given in Section 9.11.

ERROR	DESCRIPTION	COMMON CAUSE
TRAP 0	Out of memory	Forgetting to call DropMessage on item generated by function and not sent as a message.
TRAP 10	Possible multiple dispose	Using SendMsg on an input message. Using Dropmessage on an input message. Overwriting a Qdata boundary.
SYSTEMERROR #D9		Call to CkInputs has Nmin < 0.
SYSTEMERROR #DA		Call to CkInputs has Nmin > Nmax.
SYSTEMERROR #DB		Call to CkInputs has Nmax > total number of inputs for function.
SYSTEMERROR #DE		Multiple call to QSendCopyMess on the same input.

ERROR

COMMON CAUSE

SYSTEMERROR #E0

Function was not in state running when CkInputs was called.

CleanInputs returned a FALSE and still called CkInputs.

CleanInputs was not called before calling CkInputs the second time.

SYSTEMERROR #E1

Function was not in state mid_running when CleanInputs was called.

CleanInputs was called even though CkInputs had returned a NIL.

SYSTEMERROR #E9

QIILMessage, or QIILValue was called for input which does not exist.

SYSTEMERROR #EA

QIILMessage, or QIILValue was called for input which was already dealt with.

Previous call to QIILMessage, QIILValue, or QSendCopyMess.

Section AP9

APPENDICES

9.1 Using the Command Files on DEC VAX/VMS

This appendix describes the files supplied on magnetic tape to users in DEC VAX/VMS environments and provides such a user with information on the use of the files and special downloading instructions. It contains a list of the various files created by the Motorola cross-software and a listing of the code for the command file that is used to call the Motorola cross-software and name the function. The list below is a complete description of the files distributed on the magnetic tape to support the User-Written Function facility under DEC VAX/VMS:

Example Files:

BEZIER.PAS
CHCASE.PAS
COUNT.PAS
MAG.PAS
SPIRO.PAS
SPSTRUC.PAS { USERSTRUC.PAS modified to include Userdatatype }

E&S-Provided Files to Support User-Written Functions:

USERLINK.ASM
USERLINK.RO
USERSTRUC.PAS

E&S-Provided Cross-Software Command Files:

XASM.COM
XPASCAL.COM
XPASCAL2.COM
XLINK.COM
XBUILD.COM
XL.COM
XNAMES.COM

Motorola Cross-Software Files: (if purchased through E&S)

XASM.EXE
XPASCAL.EXE
XPASCAL2.EXE
XLINK.EXE
PASCALIB.RO
XASMINIT.DAT

Using the Cross-Software on VAX/VMS

Before attempting to use the cross-software, users should edit their LOGIN.COM file to invoke XNAMES.COM, which defines the necessary commands and logical names. This allows the assembler, compiler, and linker to be used exactly as described in the EXORMACS manuals. Before executing any of the commands described below, you should set your default directory to the directory containing the source files for the function you wish to compile and link. This directory should also contain copies of USERSTRUC.PAS and USERLINK.RO. As a convenience, a command file XL.COM has been provided to compile and link a user-written function, and produce the S-record file ready to download to the PS 390. All of the code must be contained in a single .PAS file, and the name of the function is assumed to be the name of the file. To invoke this command file, you should enter a command of the form:

```
$ XL <filename> <number inputs> <number outputs> <stack size>
```

If your function contains code from more than one .PAS file, or if you wish to include routines you have written in assembly language, you will have to follow these steps:

(1) Compile Pascal source files:

```
$ XPAS <filename>  
$ XPAS2 <filename>
```

(2) Assemble assembly-language source files:

```
$ XASM <filename>
```

(3) Link the object files into an S-record file:

\$ XLINK <filename1>/<filename2>../userlink,<filename1>,<filename1>;himx

(4) Add the function header line to the S-record file:

\$ XBUILD <filename> <number inputs> <number outputs> <stack size>

Special Software Installation Instructions for DEC VAX/VMS

These two steps should be taken prior to using the command files and the cross-software:

1. You should edit XNAMES.COM so that the logical names XEXEDIR and XCOMDIR reflect the actual directories where the cross-software executables and command files (respectively) have been installed.
2. You should make sure that all of the files are publicly readable.

Files Created by the Motorola Cross-Software

File Description	Contents
<filename>.PAS	Pascal source file.
<filename>.PC	P-code output from pass 1 of the Pascal compiler and input to pass 2.
<filename>.P1	Listing produced by pass 1 of the Pascal compiler.
<filename>.ASM	Assembly-language source file.
<filename>.LS	Listing file produced by assembler or pass 2 of the Pascal compiler.
<filename>.RO	Object file output by assembler of pass 2 of the Pascal compiler, input to the linker.
<filename>.SR	S-record file created by the linker.
<filename>.LL	Link map output by the linker.
<filename>.300	S-record file with header information included.

DEC VAX/VMS Command Files

This section contains the code for the DEC VAX/VMS command files that are supplied on magnetic tape by E&S to support the User-Written Function facility.

1. XASM.COM

```
$ !
$ ! Motorola 68000 Cross-Assembler
$ !
$ ! The cross-assembler may be used exactly the same as on
$ ! the EXORmacs. Note that the source file must reside in the
$ ! directory where the cross-assembler is called from, the file
$ ! extension must be .ASM, and the user must NOT specify any
$ ! other file extensions.
$ !
$ open/write il inputline.dat
$ write il ""p1""
$ close il
$ !
$ ! Now cause cross software execution
$ !
$ assign xasminit.dat INITFILE
$ copy xexedir:xasminit.dat xasminit.dat;9999
$ on error then goto finish
$ on control_y then goto finish
$ !
$ run xexedir:xasm
$ finish:
$ delete xasminit.dat;9999
$ delete inputline.dat;*
$ deassign initfile
$ exit
```

2. XBUILD.COM

```
$ ! XBUILD.COM -- build S-record .300 file for a UWF
$ !
$ ! This command file builds an S-record file for a UWF which is
```



```

$ ! ready to download to the PS 300. The header information is
$ ! found and added to the front of the output file from the linker.
$ !
$ ! You should execute this command file from the directory containing
$ ! the files for the UWF. The name of the UWF is assumed to be
$ ! the same as the name of the files. The .LL and the .SR files
$ ! produced by the linker must be present. An extension of .300 is
$ ! used for the output file.
$ ! Parameters:
$ !   p1 = name of UWF
$ !   p2 = number of function inputs
$ !   p3 = number of function outputs
$ !   p4 = estimated stack size
$ !
$ !
$ ! Ask the user for parameters if none were supplied.
$ !
$ if p1 .eqs. "" then inquire p1 "Name of UWF"
$ if p2 .eqs. "" then inquire p2 "Number of function inputs"
$ if p3 .eqs. "" then inquire p3 "Number of function outputs"
$ if p4 .eqs. "" then inquire p4 "Estimated stack size"
$ !
$ !
$ ! Determine the length of the code produced. This is found in the link
$ ! map (the .LL file). Note that the following commands depend on
$ ! knowing the exact format of the link map.
$ !
$ search /output=lsearch.tmp 'p1'.LL "Total Length"
$ open /read tmp lsearch.tmp
$ read tmp length
$ close tmp
$ delete lsearch.tmp;0
$ length := 'f$extract(24, 13, length)
$ !
$ !
$ ! Write the .300 file. This involves writing out the header, doing some
$ ! extra stuff to make sure VMS gives it the right file attributes, and
$ ! then appending the S-record file (.SR) and a semicolon.
$ !

```

```

$ open/write header fun.tmp
$ write header ""length' 'p1' 'p2' 'p3' 'p4'"
$ close header
$ assign/user fun.tmp sys$input
$ create 'p1'.300
$ append 'p1'.sr 'p1'.300
$ open/append fun 'p1'.300
$ write fun ";"
$ close fun
$ delete fun.tmp;0
$ write sys$command 'p1'.300 created"
$ exit

```

3. XL.COM

```

$ ! XL.COM -- command file to compile and link UWFs
$ !
$ ! This command file compiles and links a user-written function,
$ !   producing an S-record file ready for downloading to the PS 300.
$ !   The function must be contained in a single Pascal source file;
$ !   the name of the function is the name of the source file.
$ !   An extension of .PAS is assumed for the input file, and an
$ !   extension of .300 for the output S-record file.
$ ! You should execute this command file out of the directory
$ !   containing the Pascal source file for the user-written function.
$ !
$ ! Parameters:
$ !   p1 = name of uwf
$ !   p2 = number of inputs
$ !   p3 = number of outputs
$ !   p4 = estimated stack size
$ !
$ on error then exit
$ on control_y then exit
$ !
$ ! Compile and link the UWF, using Motorola 68000 cross-software.
$ !   You can detect compilation errors by checking for a 0-length .PC
$ !   file.
$ !

```

```

$ if p1 .eqs. "" then inquire p1 "Name of UWF"
$ xpas 'p1'                                ! pass 1 compiler
$ pcfile := 'p1'.pc                        ! check for bugs
$ if 'f$file attributes(pcfile,"ALQ") .eq. 0 -
    then goto bugs
$ xpas2 'p1'                                ! pass 2 compiler
$ xlink 'p1'/userlink,'p1','p1';himx      ! linker
$ !
$ !
$ ! Build the S-record file and clean up the extra files, leaving only
$ !   the .PAS and .300 files.
$ !
$ xbuild 'p1' 'p2' 'p3' 'p4'
$ delete 'p1'.ll;*, 'p1'.pl;*, 'p1'.ro;*, 'p1'.sr;*, 'p1'.pc;*
$ exit
$ !
$ ! Clean up after compilation errors, leaving the .PL file so the user
$ !   can find his bugs.
$ !
$ bugs:
$ write sys$output "Aborted -- compilation errors"
$ delete 'p1'.pc;*
$ exit

```

4. XLINK.COM

```

$ !
$ ! Motorola 68000 Cross-Linker
$ !
$ ! The linker may be used exactly the same as on the EXORmacs.
$ ! Note that all files must reside in the directory where the linker
$ ! is called from, all files must be .RO, and the user must NOT
$ ! specify the file extension. Also, all output files must be explicitly
$ ! specified.
$ !
$ open/write il inputline.dat
$ write il ""p1""
$ close il
$ !

```

```

$ ! Now cause cross software execution.
$ !
$ copy xexedir:pascalib.ro pascalib.ro;9999
$ on error then goto finish
$ on control_y then goto finish
$ assign/user mode sys$command sys$input
$ run xexedir:xlink
$ !
$ finish:
$ delete inputline.dat;*
$ delete pascalib.ro;9999
$ delete headerf.dat;*
$ exit

```

5. XNAMES.COM

```

$ ! XNAMES.COM -- set up logical names and symbols for using cross
$ !   software.
$ !
$ ! You should execute this command file (as in your LOGIN.COM)
$ !   before attempting to use the Motorola 68000 cross-software to
$ !   build user-written functions.
$ !
$ !
$ ! Define logical names for the actual locations of the cross-software
$ !   command files and executables. These should be updated during
$ !   installation as necessary.
$ !
$ assign disk$ias soft:[loosemore.uwf.dist.com]      xcomdir
$ assign disk$ias soft:[loosemore.uwf.dist.exe]      xexedir
$ !
$ !
$ ! The following aliases allow the Motorola cross-software to be used
$ !   under VMS exactly as described in the EXORMacs manuals.
$ !
$ xasm   ::= @xcomdir:xasm           ! invoke cross-assembler
$ xpas   ::= @xcomdir:xpascal        ! invoke cross-compiler, pass 1
$ xpas2  ::= @xcomdir:xpascal2       ! invoke cross-compiler, pass 2
$ xlink  ::= @xcomdir:xlink          ! invoke cross-linker

```

```

$ !
$ !
$ ! Finally, two more command files to build S-record files ready to
$ ! download to the PS 300.
$ !
$ xl      := @xcomdir:xl    ! compile, link, and build S-record file
$ xbuild := @xcomdir:xbuild ! build S-record file
$ exit

```

6. XPASCAL.COM

```

$ !
$ ! Motorola 68000 Pascal Cross-Compiler
$ !
$ ! The compiler may be used exactly the same as on the EXORmacs.
$ ! Note that the source file must reside in the directory where the
$ ! compiler is called from, the file extension must be .PAS, and the
$ ! user must NOT specify any other file extensions.
$ !
$ open/write il inputline.dat $ write il ""p1""
$ close il
$ !
$ on error then goto finish
$ on control_y then goto finish
$ !
$ ! Now cause cross software execution.
$ !
$ run xexedir:xpascal
$ finish:
$ delete inputline.dat;*
$ exit

```

7. XPASCAL2.COM

```

$ !
$ ! Motorola 68000 Pascal Cross-Compiler (Phase 2)
$ !
$ ! The compiler may be used exactly the same as on the EXORmacs.

```

```
$ ! Note that the P-code file must reside in the directory where the
$ ! compiler is called from, the file extension must be .PC, and the
$ ! user must NOT specify any other file extensions.
$ !
$ open/write il inputline.dat
$ write il ""p1""
$ close il
$ !
$ on error then goto finish
$ on control_y then goto finish
$ !
$ ! Now cause cross software execution
$ ! $ run xexedir:xpascal2
$ finish:
$ delete inputline.dat;*
$ exit
```

9.2 Using the Command Files on DEC VAX/UNIX

This section describes the files supplied on magnetic tape to users in DEC VAX/UNIX environments and provides such a user with information on the use of the files and special downloading instructions. The list below is a complete description of the files distributed on the magnetic tape to support the User-Written Function facility under DEC VAX/UNIX:

Example Files:

bezier.pas
chcase.pas
count.pas
mag.pas
spiro.pas
spstruc.pas { USERSTRUC.PAS modified to include Userdatatype }

E&S-Provided Files to Support User-Written Functions:

userlink.asm
userlink.ro
userstruc.pas

E&S-Provided Cross-Software Command Files

xbuild
xl
xnames

Motorola Cross-Software Files (if purchased through E&S)

uxasm
uxpascal
uxpascal2
uxlink
pascalib.ro
asminit.dat

Using the Cross-Software on Unix 4.2 BSD

Before attempting to use the cross-software, users should edit their .cshrc file to “source” the file xnames, which defines the necessary aliases and shell variables. This allows the assembler, compiler, and linker to be used

as described in the EXORMACS manuals. (The only exception is that multiple input files should be separated by “+” instead of “/”, and if options are specified using “;”, the entire argument list should be quoted.)

Before executing any of the commands described below, you should set your working directory to the directory containing the source files for the function you wish to compile and link. This directory should also contain copies of `userstruc.pas` and `userlink.ro`. Since Unix is case sensitive, you must remember to use consistent case for filenames.

As a convenience, a shell script `xl` has been provided to compile and link a user-written function, and produce the S-record file ready to download to the PS 390. All of the code must be contained in a single `.pas` file, and the name of the function is assumed to be the name of the file. To invoke this shell script, you should enter a command of the form:

```
% xl <filename> <number inputs> <number outputs> <stack size>
```

If your function contains code from more than one `.pas` file, or if you wish to include routines you have written in assembly language, you will have to follow these steps:

(1) Compile Pascal source files:

```
% xpas <filename>  
% xpas2 <filename>
```

(2) Assemble assembly-language source files:

```
% xasm <filename>
```

(3) Link the object files into an S-record file:

```
% xlink '<filename1>+<filename2>...+userlink,<filename1>,<filename1>;himx'
```

(4) Add the function header line to the S-record file:

```
% xbuild <filename> <number inputs> <number outputs> <stack size>
```

Special Software Installation Instructions for DEC VAX/UNIX

These steps should be taken prior to using the commands files and the cross-software:

1. You should edit the shell script **xnames** so that the shell variables **\$xexedir** and **\$xcomdir** reflect the actual pathnames of the directories where the cross-software executables and shell scripts (respectively) have been installed.
2. You must also make sure that the files **pascalib.ro** and **asminit.dat** can be found in:

```
/usr/local/lib/pas68/pascalib.ro and
/usr/local/lib/pas68/asminit.dat
```

respectively. This may be done either by copying the files, or by creating a link to the files.

3. Make sure that all of the files are publicly readable.

Files Created by the Motorola Cross-Software

File Description	Contents
<filename>.pas	Pascal source file.
<filename>.pc	P-code output from pass 1 of the Pascal compiler and input to pass 2.
<filename>.pl	Listing produced by pass 1 of the Pascal compiler.
<filename>.asm	Assembly-language source file.
<filename>.ls	Listing file produced by assembler or pass 2 of the Pascal compiler.
<filename>.ro	Object file output by assembler of pass 2 of the Pascal compiler, input to the linker.
<filename>.sr	S-record file created by the linker.
<filename>.ll	Link map output by the linker.
<filename>.300	S-record file with header information included.

DEC VAX/UNIX Command Files

This section contains the code for the DEC VAX/UNIX command files that are supplied on magnetic tape by E&S to support the User-Written Function facility.

1. xnames

```
# xnames -- set up names for using 68k cross-software
#
# You should "source" this file in your .cshrc file before attempting
#   to use the Motorola 68000 cross-software to build
#   user-written functions
#
# Define names for the actual locations of the cross software shell
#   scripts and executables.  These should be updated during
#   installation as necessary.
#
set xexedir=-loosemor/dist/exe
set xcomdir=-loosemor/dist/com
#
#
# The following aliases allow the Motorola cross-software to be used
#   under Unix exactly as described in the EXORMACS manuals.
# The only exceptions are:
# (1) Multiple input files should be separated with a "+" instead of "/".
# (2) If you specify options using ";", the entire parameter list should
#     be enclosed in quotes.
#
alias xasm      $xexedir/uxasm
alias xlink     $xexedir/uxlink
alias xpas      $xexedir/uxpascal
alias xpas2     $xexedir/uxpascal2
#
#
# Finally, two more command files to build S-record files ready to
#   download to the PS 300:
#
alias xl        csh $xcomdir/xl
alias xbuild    csh $xcomdir/xbuild
```

2. xl

```
# xl -- compile and link user-written functions, producing S-record file
#
# This shell script compiles and links a user-written function, producing
# an S-record file ready for downloading to the PS 300. The function
# must be contained in a single Pascal source file; the name of the
# function is the name of the source file. An extension of .pas is
# assumed for the input file, and an extension of .300 for the output
# S-record file.
# You should execute this shell script out of the directory containing the
# Pascal source file for the user-written function.
#
# Parameters:
# $1 = name of UWF
# $2 = number of inputs
# $3 = number of outputs
# $4 = estimated stack size
#
#
# Get the name of the function (required).
#
if ($#argv > 0) then
    set name=$1
else
    echo -n 'Name of UWF: '
    set name=($<)
endif
#
#
# Compile and link the UWF, using Motorola 68000 cross-software. We
# can detect compilation errors by checking for a 0-length .pc file.
#
xpas $name
set pcrlen='ls -l $name.pc | awk '{ print $4 }''
if ($pcrlen == 0) goto bugs
```

```

xpas2 $name
xlink "$name+userlink,$name,$name;himx"
#
# Build the S-record file and clean up the extra files, leaving only the
#   .pas and the .300 files.
#
xbuild $name $2 $3 $4
rm $name.ll $name.pl $name.ro $name.sr $name.pc
exit
#
#
# Clean up after compilation errors, leaving the .pl file so the user
#   can find his bugs.
#
bugs: echo 'Aborted -- compilation errors'
rm $name.pc
exit

```

3. xbuild

```

# xbuild -- build S-record .300 file for a UWF
#
# This command file builds an S-record file for a UWF which is ready to
#   download to the PS 300. The header information is found and added
#   to the front of the output file from the linker.
# You should execute this shell script from the directory containing the
#   files for the UWF. The name of the UWF is assumed to be the same
#   as the name of the files. The .ll and the .sr files produced by the
#   linker must be present. An extension of .300 is used for the output
#   file.
# Parameters:
#   $1 = name of UWF
#   $2 = number of function inputs
#   $3 = number of function outputs
#   $4 = estimated stack size
#
# Ask the user for parameters if none were supplied.
#
if ($#argv > 0) then

```

```

        set name=$1
    else
        echo -n 'Name of UWF: '
        set name=($<)
    endif
    if ($#argv > 1) then
        set inputs=$2 else
        echo -n 'Number of function inputs: '
        set inputs=($<)
    endif
    if ($#argv > 2) then
        set outputs=$3
    else
        echo -n 'Number of function outputs: '
        set outputs=($<)
    endif
    if ($#argv > 3) then
        set stacksize=$4
    else
        echo -n 'Estimated stack size: '
        set stacksize=($<)
    endif
    #
    #
    # Determine the length of the code produced. This is found in the link
    # map (the .ll file).
    # set len='awk '/Total Length/ { print $4 }' $name.ll'
    #
    #
    # Write the .300 file. This involves putting together the header line and
    # appending a semicolon to the end of the file.
    #
    echo $len $name $inputs $outputs $stacksize >$name.300
    cat $name.sr >>$name.300
    echo ';' >>$name.300
    echo $name.300 created
    exit

```

9.3 Using the Cross-Software on IBM VM/SP

This section describes the files supplied on magnetic tape to users in IBM VM/SP environments and provides such a user with information on the use of the Motorola cross-software. The list below is a complete description of the files distributed on the magnetic tape to support the User-Written Function facility under IBM VM/SP:

Example Files:

BEZIER PASCAL
CHCASE PASCAL
COUNT PASCAL
MAG PASCAL
SPIRO PASCAL
SPSTRUC PASCAL {USERSTRUC.PAS modified to include Userdatatype}

E&S-Provided Files to Support User-Written Functions:

USERLINK ASSEMBLE
USERLINK OBJECT
USERSTRU PASCAL

Motorola Cross-Software Files (if purchased through E&S)

ASMB MODULE
PASCALCO MODULE
DIRECT MODULE
LINK MODULE
PASCALIB DATA
ASMINIT DATA

How To Use the Cross-Software on IBM VM/SP

It is very important that you are familiar with the following information before you try to use the Motorola cross-software. Because of the nature of the IBM environment, explicit files that call the cross-software are not provided by E&S. Use the following information to create the S-record file that contains the code for your user-written function, correctly format the file for downloading, and download it to the PS 390.

Pascal Differences

The IBM version of the Motorola Pascal cross-compiler uses different lexical conventions than standard Pascal. In particular, you should:

- use @ instead of ↑ for pointer references
- use (. and .) instead of [] for array references.

You cannot refer to files by name explicitly in \$F=<filename> statements. To include files during compilation, the \$F statement should refer to a DDNAME. You must include a FILEDEF command to define that DDNAME prior to invoking the cross-compiler.

Using the Cross-Compiler

Before invoking the cross-compiler, you must execute a number of FILEDEF commands to define the files used. These files are:

SOURCE The file containing the Pascal source code to be compiled. This file is read in by pass 1 of the compiler.

LISTING The listing output by pass 1 of the compiler.

PCODE This file contains the intermediate code produced by pass 1 of the compiler and used as input by pass 2.

OUTPUT Both passes of the compiler write results of the compilation to the file OUTPUT, which is normally associated with the terminal.

P2LIST The listing file output by pass 2 of the compiler.

FILE1 Pass 2 writes the relocatable object module to this file.

FILE2 This is a temporary file used by pass 2 of the compiler.

In addition, if you have referenced any files to be included via \$F statements in your Pascal source file, you must also execute FILEDEF commands for these files.

The following exec file, UWFPASC EXEC, will compile the Pascal source file input as the first parameter. It is assumed that the source file includes USERSTRU PASCAL through a statement of the form:

```

{$F=INCLUDE }

FILE:  UWFPASC  EXEC

&TRACE ERR
&CONTROL &OFF
FILEDEF  *  CLEAR
FILEDEF  OUTPUT TERMINAL (RECFM F LRECL 80 BLOCK 80
FILEDEF  SOURCE DISK &1 PASCAL A
FILEDEF  INCLUDE DISK USERSTRUC PASCAL A
FILEDEF  LISTING DISK &1 LISTING A (RECFM VBA LRECL 133 BLOCK 3990
FILEDEF  PCODE DISK PASCPCOD DATA A (RECFM VB LRECL 256 BLOCK 2600
FILEDEF  P2LIST DISK &1 DATA A (RECFM VBA LRECL 133 BLOCK 3990
FILEDEF  FILE1 DISK &1 OBJECT A (RECFM FB LRECL 256 BLOCK 2560
FILEDEF  FILE2 DISK PASCFIL2 DATA (RECFM FB LRECL 256 BLOCK 2560
PASCALCO
DIRECT
EXIT

```

After invoking the cross-compiler, you should check the LISTING file for errors.

Using the Cross-Assembler

The cross assembler requires that you execute FILEDEF statements to define the following DDNAMES:

OUTPUT	Normally, this file is allocated to the terminal.
SOURCE	This should be allocated to the assembly source input file.
LISTING	The cross assembler will write its output listing to this file.
OBJECT	The object code output by the assembler will be written to this file.
INITFILE	The initialization file, which must be read in at the beginning of each invocation of the cross-assembler.

The following exec file, MYASMY EXEC, will assemble the source file passed as parameter 1:

```

FILE:  MYASMY  EXEC

& TRACE ON
& CONTROL &OFF

```



```

FILEDEF * CLEAR
FILEDEF OUTPUT TERMINAL
FILEDEF SOURCE DISK &1 ASSEMBLE * (RECFM FB LRECL 80 BLOCK 3200
FILEDEF LISTING DISK &1 LISTING * (RECFM VB LRECL 133 BLOCK 3990
FILEDEF OBJECT DISK &1 OBJECT * (RECFM FB LRECL 256 BLOCK 2560
FILEDEF INITFILE DISK ASMINIT DATA C1 (RECFM FB LRECL 80 BLOCK 3200
ASMB
& EXIT

```

Linking

Before invoking the linker, you should execute FILEDEF statements to define the following files:

INPUT	The file containing the linker commands. This may be assigned to the terminal.
OUTPUT	The map file output by the linker.
OUTFIL	The load module produced as output by the linker.
HEADERF	A temporary file used by the linker for processing the H option.
PASCALIB	The default run-time library for Pascal object modules.

In addition, you must execute a FILEDEF for each object file you wish to input to the linker. These are referenced by the file INPUT, which should contain commands of the form:

```

INPUT <ddname1>
INPUT <ddname2>
.
.
END

```

For example, the file LINK TXT contains the following commands:

```

INPUT OBJ1
INPUT OBJ2
END

```

This file is referenced by UWFLINK EXEC. This exec file takes the name of a single object file as a parameter, and links it with USERLINK OBJECT.

```
FILE: UWFLINK EXEC
```

```
&TRACE ON
&CONTROL &OFF
FILEDEF * CLEAR
FILEDEF INPUT DISK LINK TXT A (RECFM F LRECL 80 BLOCK 80
FILEDEF OUTPUT DISK &1 MAP A (RECFM F LRECL 80 BLOCK 80
FILEDEF OUTFIL DISK &1 LOAD A (RECFM VB LRECL 256 BLOCK 2600
FILEDEF HEADERF DISK M68KHDRF DATA C (REDFM F LRECL 80 BLOCK 80
FI OBJ1 DISK &1 OBJECT A (RECRM FM LRECL 256 BLOCK 2560
FI OBJ2 DISK USERLINK OBJECT A (RECFM FB LRECL 256 BLOCK 2560
FILEDEF PASCALIB DISK PASCALIB DATA C1
DESBUF
LINK
```

Modifying the S-Record File

Before you can download the S-record file for the UWF to the PS 390, you must modify it to contain a header line of the format described in section 9.7, and terminate the file with a semicolon.

You should examine the map file output by the linker to determine the length of code for including in the header line.

Downloading the UWF to the PS 390

The following is an example of a command file to run the program SRecsnd that sends the specified file to the PS 390 (this requires the file name as a parameter):

```
&TRACE ON
&CONTROL &OFF
EXEC P6P FILEDEF * CLEAR
FILEDEF INPUT TERMINAL
FILEDEF OUTPUT TERMINAL
FILEDEF SRECFILE DISK &1 LOAD A (RECFM VB LRECL 256 BLOCK 2600
LOAD SRECSND (START
```

This is an example of the Pascal program, SRecsnd, that sends a file to PS 390. This program makes calls to the PS 390 GSR routines.

```
FILE: SRECSND PASCAL *

program srecsnd (input, output, srecfile );

CONST
```

```

%INCLUDE PROCONST

TYPE
    %INCLUDE PROTYPES

VAR
    Ssrecfile  : text;
    istr       : string ( 256 );
    crlfa      : packed array ( . 1..2 .) of char;
    crlf       : string( 2 );

%INCLUDE PROEXTRN
PROCEDURE err ( errnum : integer );
    BEGIN
        writeln( ' got error: ', errnum );
        END;

    BEGIN
        pattach( 'junk', err );
        reset( srecfile );
        crlfa ( . 1 .) := CHAR ( 13 );
        crlf  := STR( crlfa );
        pmuxg ( 7, err );
        WHILE NOT EOF (srecfile) DO
            BEGIN readln (srecfile, istr );
                Pputgx (istr, err );
                Pputgx (crlf, err );
            END;
        writeln;
        PDetach ( err );
        END.

```

9.4 Using the Command Files on IBM MVS/TSO

The list below is a complete description of the files distributed on the magnetic tape to support the User-Written Function facility under IBM MVS/TSO:

Example Files:

BEZIER PASCAL
CHCASE PASCAL
COUNT PASCAL
MAG PASCAL
SPIRO PASCAL
SPSTRUC PASCAL {USERSTRUC.PAS modified to include Userdatatype}

E&S-Provided Files to Support User-Written Functions:

USERLINK ASSEMBLE
USERLINK OBJECT
USERSTRU PASCAL

Motorola Cross-Software Files (if purchased through E&S)

ASMB MODULE
PASCALCO MODULE
DIRECT MODULE
LINK MODULE
PASCALIB DATA
ASMINIT DATA

The MVS/TSO user should refer to the Motorola manuals distributed with the Motorola cross-software for instructions on the use of the cross-compiler, cross-assembler, and cross-linker.

For information on preparing the file for downloading, and for downloading it to the PS 390, refer to the sections entitled “Modifying the S-Record File” and “Downloading the UWF to the PS 390” in section 9.3 of this manual.

9.5 USERSTRUC.PAS

This section contains the examples and supplied command files from the USERSTRUC.PAS file, that is distributed on magnetic tape. Reference to this file is made throughout the document and it is provided here for completeness.

```
CONST   MaxInputQueues = 127; { Max #of input queues for a function }

TYPE Int16 = -32768..32767; { 16-bit integer }
      Int8 = -128..127;     { 8-bit integer }
      PtrQdata = ↑Qdata ; { pointer to a message }

      PtrUWFInQarray = ↑UWFInQarray;
      UWFInQarray = Array [1..MaxInputQueues] of PtrQdata;
      InUWFQarray = UWFInQarray; {for compatibility with older
                                   versions}

double =
  RECORD
    c: Int16;           { 16 bit biased binary exponent}
    m: integer;         { 32 bit floating point fraction}
    notused: int16;      { waste, to make = 8 bytes for}
  END;                  { faster array indexing}
PS 300_floating_point = double; { old name, for compatibility}

Vector = ARRAY [ 0..3 ] OF double;
Matrix = ARRAY [ 0..3 , 0..3 ] OF double;

Bytespell = ARRAY [ 1..255 ] OF char;

Qdtype = { types of Qdata (message) blocks }
(
  { 0} Qreset,    { dataless: reset a function instance           }
  { 1} Qprompt,   { dataless: flush the CI pipeline               }
  { 2} Qboolean,  { normal carrier of boolean values              }
  { 3} Qinteger,  { normal carrier of integer values              }
  { 4} Qreal,     { normal carrier of floating point values        }
  { 5} Qstring,   { original carrier of byte strings              }
  { 6} Qpacket,   { new carrier of byte strings                   }
  { 7} Qmorepacket, { alternate to Qpacket (with the distinction  }
                  { making a difference only on the link          }
                  { between F:DEPACKET and F:DEMUX/F:CIRROUTE).   }
  { 8} Qmove2,    { 2D vector including P bit                     }
  { 9} Qdraw2,    { 2D vector including L bit                     }
  {10} Qvec2,     { 2D vector with no P/L bit (normal vector)      }
  {11} Qmove3,    { 3D vector including P bit                     }
)
```

```

{12} Qdraw3, { 3D vector including L bit }
{13} Qvec3, { 3D vector with no P/L bit (normal vector) }
{14} Qmove4, { 4D vector including P bit }
{15} Qdraw4, { 4D vector including L bit }
{16} Qvec4, { 4D vector with no P/L bit (normal vector) }
{17} Qmat2, { 2x2 matrix }
{18} Qmat3, { 3x3 matrix }
{19} Qmat4, { 4x4 matrix }
{20} Qusertype, { type which user may use to define own message }

```

```

{ padding, to make the field 16-bit, as it is in the full system }

```

```

Pad, Pae, Paf, Pag, Pah, Pai, Paj, Pak, Pal, Pam,
Pbd, Pbe, Pbf, Pbg, Pbh, Pbi, Pbj, Pbk, Pbl, Pbm,
Pcd, Pce, Pcf, Pcg, Pch, Pci, Pcj, Pck, Pcl, Pcm,
Pdd, Pde, Pdf, Pdg, Pdh, Pdi, Pdj, Pdk, Pdl, Pdm,
Ped, Pee, Pef, Peg, Peh, Pei, Pej, Pek, Pel, Pem,
Pfd, Pfe, Pff, Pfg, Pfh, Pfi, Pfj, Pfk, Pfl, Pfm,
Pgd, Pge, Pgf, Pgg, Pgh, Pgi, Pgj, Pgk, Pgl, Pgm,
Phd, Phe, Phf, Phg, Phh, Phi, Phj, Phk, Phl, Phm,
Pid, Pie, Pif, Pig, Pih, Pii, Pij, Pik, Pil, Pim,
Pjd, Pje, Pjf, Pjg, Pjh, Pji, Pjj, Pjk, Pjl, Pjm,
Pkd, Pke, Pkf, Pkg, Pkh, Pki, Pkj, Pkk, Pkl, Pkm,
Pld, Ple, Plf, Plg, Plh, Pli, Plj, Plk, Pll, Plm,
Pmd, Pme, Pmf, Pmg, Pmh, Pmi, Pmj, Pmk, Pml, Pmm,
Pnd, Pne, Pnf, Png, Pnh, Pni, Pnj, Pnk, Pnl, Pnm,
Pod, Poe, Pof, Pog, Poh, Poi, Poj, Pok, Pol, Pom,
Ppd, Ppe, Ppf, Ppg, Pph, Ppi, Ppj, Ppk, Ppl, Ppm,
Pqd, Pqe, Pqf, Pqg, Pqh, Pqi, Pqj, Pqk, Pql, Pqm,
Prd, Pre, Prf, Prg, Prh, Pri, Prj, Prk, Prl, Prm,
Psd, Pse, Psf, Psg, Psh, Psi, Psj, Psk, Psl, Psm,
Ptd, Pte, Ptf, Ptg, Pth, Pti, Ptj, Ptk, Ptl, Ptm,
Pud, Pue, Puf, Pug, Puh, Pui, Puj, Puk, Pul, Pum,
Pvd, Pve, Pvf, Pvg, Pvh, Pvi, Pvj, Pvk, Pvl, Pvm,
Pwd, Pwe, Pwf, Pwg, Pwh, Pwi, Pwj, Pwk, Pwl, Pwm,
Pxd, Pxe, Pxf, Pxg, Pxh, Pxi, Pxj, Pxk, Pxl, Pxm,
Pyd, Pye, Pyf, Pyg, Pyh, Pyi, Pyj, Pyk, Pyl, Pym,
Pzd, Pze, Pzf, Pzg, Pzh, Pzi, Pzj, Pzk, Pzl, Pzm
) ;

```

```

{ TYPE declarations continued }

```

```

Qdata =
RECORD
  Next: Ptrqdata ; { next message in a list of messages }
  CASE Qtyp: Qdtype OF { type of message }
    { Qreset: no datum carried }
    { Qprompt: no datum carried }

```

```

        Qboolean:
        (
b: boolean
        ) ;
        Qinteger:
        (
i: integer
        ) ;
        Qreal:
        (
r: double
        ) ;
        Qstring: { an old form of byte-string message }
        (
l: int16 ;                { # bytes of message }
Qs_pad: int16 ;           { padding ... aligns with Qpacket }
n: Bytespell             { bytes of message }
        ) ;
        Qpacket, Qmorepacket: { newer form of byte-string }
        (
P_lth: int16 ;            { max byte number }
P_beg: int16 ;            { min byte number }
P_cnt: Bytespell         { bytes of message }
        ) ;
        Qmove2, Qdraw2, Qvec2,
        Qmove3, Qdraw3, Qvec3,
        Qmove4, Qdraw4, Qvec4:
        (
V4: Vector                { all vectors use 4D indexing }
        ) ;
        Qmat2, Qmat3, Qmat4:
        (
            Mat4: Matrix    { all matrices use 4x4 indexing }
        ) ;
        END ; { Qdata }

{ **** Note: there are no global VARs available **** }

FUNCTION CkPrivate : Ptrqdata;
    FORWARD ;
PROCEDURE SavePrivate ( msg : Ptrqdata );
    FORWARD ;
FUNCTION CkInputs ( first, last : Int16 ) : PtrUWFInQarray;
    FORWARD ;
FUNCTION CleanInputs : BOOLEAN;
    FORWARD ;

```

```

PROCEDURE SendMsg ( VAR msg : Ptrqdata; outport : Int16 );
    FORWARD ;
PROCEDURE QSendCopyMsg ( source, destination : Int16 );
    FORWARD ;
PROCEDURE QIillMessage ( input : Int16 );
    FORWARD ;
PROCEDURE QIillValue ( input : Int16 );
    FORWARD ;
PROCEDURE Qincompatmsgs ( one : Int16; theother : Int16 );
    FORWARD ;
FUNCTION Msgcopy(m: Ptrqdata): Ptrqdata ;
    FORWARD ;
PROCEDURE Dropmessage(VAR m: Ptrqdata) ;
    FORWARD ;
PROCEDURE Systemerror(n: Int16) ;
    FORWARD ;

FUNCTION Fpecomp(VAR X1,X2: double): Int8 ;
    FORWARD ;
PROCEDURE Fpabs(VAR r: double) ;
    FORWARD ;
PROCEDURE FCadd(VAR Augend, Addend: double;
                VAR Sum: double) ;
    FORWARD ;
PROCEDURE FCsubtract(VAR Minuend, Subtrahend: double;
                    VAR Difference: double) ;
    FORWARD ;
PROCEDURE FCmultiply(VAR a, b: double;
                    VAR Product: double) ;
    FORWARD ;
PROCEDURE FCp2multiply(VAR Innum: double; Power: integer;
                    VAR Outnum: double) ;
    FORWARD ;
PROCEDURE FCdivide(VAR Dividend, Divisor: double;
                    VAR Quotient: double) ;
    FORWARD ;
PROCEDURE FCint2double( num : Integer; VAR Floated: double) ;
    FORWARD ;
PROCEDURE FCround(VAR Innum: double; VAR Outnum: integer) ;
    FORWARD ;
PROCEDURE FCinteger(VAR Innum: double; VAR Outnum: integer) ;
    FORWARD ;
FUNCTION FCnearzero ( VAR tiny : double; negpower2 : int16
                    ) : int8 ;

    FORWARD ;{ negpower2=1 --> within .5; =2 --> within .25 }

```



```

PROCEDURE FCsqroot(VAR a: double; VAR Sqroot: double) ;
    FORWARD ;

PROCEDURE Sincos(Angle: integer; VAR Sine: double;
    VAR Cosine: double) ;
    FORWARD ;

FUNCTION Rndmnumber(seed : Integer): Int16;
    FORWARD ;

FUNCTION Newqpacket( Typ: Qdtype; { Qpacket or Qmorepacket }
    Nbytes: Int16): Ptrqdata ;
    FORWARD ;

FUNCTION Newqreal: Ptrqdata ;
    FORWARD ;

FUNCTION Newqinteger: Ptrqdata ;
    FORWARD ;

FUNCTION Newqboolean: Ptrqdata ;
    FORWARD ;

FUNCTION Newqnil(Typ: Qdtype): Ptrqdata ; { Qreset; Qprompt }
    FORWARD ;

FUNCTION Newqvector(Typ: Qdtype): Ptrqdata ; { Qvec2, ... }
    FORWARD ;

FUNCTION Newqmatrix(Typ: Qdtype): Ptrqdata ; { Qmat2, ... }
    FORWARD ;

FUNCTION Vfetch( Name: Ptrqdata) : Ptrqdata; { a Qpacket }
    FORWARD ;

PROCEDURE Vstore( Name: Ptrqdata; VAR New_val: Ptrqdata) ;
    FORWARD ;

FUNCTION My_name : Ptrqdata ;
    FORWARD ;

PROCEDURE My_in_out ( VAR N_in,N_out: int16 ) ;
    FORWARD ;

FUNCTION Ticks: integer ;
    FORWARD ;

FUNCTION Csecs: integer ;
    FORWARD ;

FUNCTION Frames: integer ;
    FORWARD ;

PROCEDURE Hrtime(VAR c,f,d: integer) ;
    FORWARD ;

PROCEDURE Char_text(c: char; VAR b,e: Int16; VAR Ca: Bytespell) ;
    FORWARD ;

PROCEDURE Text_text(VAR B1,E1: Int16; VAR Ca1: Bytespell;
    VAR B2,E2: Int16; VAR Ca2: Bytespell) ;
    FORWARD ;

```

```

PROCEDURE Int_text(n: integer; Ns,Nz: Int16;
                  VAR b,e: Int16; VAR Ca: Bytespell) ;
    FORWARD ;
PROCEDURE Time_text(n: integer; VAR b,e: Int16; VAR Ca: Bytespell) ;
    FORWARD ;
PROCEDURE Real_text(VAR r: double; VAR b,e: Int16; VAR Ca: Bytespell)
    FORWARD ;

FUNCTION NewTry ( num_bytes : INTEGER ) : Ptrqdata;
    FORWARD;
PROCEDURE UWFEError (VAR msg : Ptrqdata );
    FORWARD;
PROCEDURE Set_Cness( input : Int16; cotype: Boolean );
    FORWARD;

```

Advanced UWF Procedures

```

FUNCTION Lk_cursuffix ( Nlth: integer; VAR Nspell: Namespell) :
    Ptralphabl;
    FORWARD;
FUNCTION Lk_nosuffix ( length: Integer; Cinum: Int 8; suffix: Char;
    VAR Nspell:Namespell) : Ptralphabl;
    FORWARD;
PROCEDURE Lgaupdate ( Name: Ptralphabl; data: Ptrnamedentity;
    VAR Uph,Upt: Ptravupblk);
    FORWARD;
PROCEDURE Announceupdate ( VAR Uph,Upt: Ptravupblk);
    FORWARD;
PROCEDURE Msgstore ( VAR Uph,Upt: Ptravupblk);
    FORWARD;
PROCEDURE Msgstore ( Msg: Ptrqdata; a: Ptralphabl; n: integer);
    FORWARD;
PROCEDURE Setlock ( VAR x: Lock);
    FORWARD;
PROCEDURE Clrlock ( VAR s: Lock);
    FORWARD;
PROCEDURE Incausage ( a: Ptralphabl);
    FORWARD;
PROCEDURE Decausage ( a: Ptralphabl);
    FORWARD;
PROCEDURE AcpProof ( VAR location: ptracpcblk; newval: ptracpcblk);
    FORWARD;
PROCEDURE Acpprf1 ( VAR location: ptrsavstate; value: ptrsavstate);
    FORWARD;
PROCEDURE OLBaddtset ( A_son: Ptralphabl; A_father: Ptralphabl;
    VAR Uph,Upat: Ptravupblk; VAR Error: boolean; Optimize :
    boolean);
    FORWARD;

```

```

PROCEDURE Removefromset ( A_father: Ptralphablk; A_son: Ptralphablk;
    VAR Uph,Upt: Ptravupblk; VAR Error: boolean);
    FORWARD;
FUNCTION FetchBlock ( block: PtrNamedentity; Upt: Ptravupblk):
    Ptrnamedentity;
    FORWARD;
PROCEDURE Acp_v3f ( VAR v: Vector; VAR acpv: Vec3f; pl: boolean);
    FORWARD;
PROCEDURE Acp_v2f ( VAR v: Vector; VAR acpv: Vec2f; pl: boolean);
    FORWARD;
PROCEDURE Acp_v3b ( VAR v: Vector; VAR acpv: Vec3b; pl: boolean; exp:
    Int16);
    FORWARD;
PROCEDURE Acp_v2b ( VAR v: Vector; VAR acpv: Vec2b; pl: boolean; exp:
    Int16);
    FORWARD;
PROCEDURE NStoreVector ( VAR block: Ptrnamedentity; VAR v: Vector;
    pl: boolean firstblock: Ptrnamedentity);
    FORWARD;
FUNCTION nNewAcpdata ( n: int16; t: Dattype): Ptrnamedentity;
    FORWARD;
PROCEDURE Store3x3 ( VAR m: Matrix3; Ob: Ptrnamedentity; n: int16);
    FORWARD;
PROCEDURE Store4x4 ( VAR m: Matrix4; Ob: Ptrnamedentity; n: int16);
    FORWARD;
PROCEDURE Drop_name ( a: Ptralphablk);
    FORWARD;
PROCEDURE GetVector ( block: Ptrnamedentity; index: Int16;
    VAR v: Vector; VAR pl: Boolean);
    FORWARD;
PROCEDURE Rawbacopy ( Nbytes: int16; VAR Inba,Outba: int8);
    FORWARD;
PROCEDURE Rawcbcopy ( Nbytes: int16; VAR Inba:char; VAR Outba: int8);
    FORWARD;
PROCEDURE Rawchcopy ( Nbytes: int16; VAR Inba,Outba: char);
    FORWARD;
FUNCTION Size_of ( b: Ptrnamedentity): integer;
    FORWARD;
FUNCTION FetchAdnum ( f: Ptrnamedentity; upt: Ptravupblk): Int16;
    FORWARD;
FUNCTION nFetchCopy ( block: PtrNamedentity; start: Int16; count:
    Int16; upt: Ptravupblk) : PtrNamedentity;
    FORWARD;
PROCEDURE WaitFrame ( framecnt : integer);
    FORWARD;
FUNCTION loc_thead : Ptrcommhead;
    FORWARD;

```

```

FUNCTION ptr_dcb : Ptrdcb;
    FORWARD;
PROCEDURE DropNE ( f: Ptrnamedentity);
    FORWARD;
PROCEDURE Newreturns ( givemenil : boolean);
    FORWARD;
PROCEDURE Reactivate ;
    FORWARD;
FUNCTION Myanyoutputs ( n: int8): Boolean;
    FORWARD;
PROCEDURE Pushmyinput ( m: Ptrqdata; n: Int8);
    FORWARD;
PROCEDURE WaitCsec ( csecnt : integer);
    FORWARD;

```

9.6 Function Header Line Format

This section contains a description of the function header line format used to name a function, define the number of inputs and outputs, and provide the stack usage.

The function-naming command must use the following syntax:

`<length> <function_name> <number inputs> <number outputs> <stack size>`

where:

<length> is the number of bytes in decimal of the file (the number of bytes can be found in the linker listing labeled 'Total Length').

<function_name> is the PS 390 name for the user-written function.

<number inputs> is the number of input queues of the user-written function.

<number outputs> is the number of output ports of the user-written function.

<stack size> is the estimated total stack usage requirements in decimal. (Refer to Section AP8.6 for estimates of stack usage of some of the utility procedures.)

These parameters are delimited by spaces.

9.7 S-Record Format

The S-record format for modules was devised for the purpose of encoding programs or data files for transportation between computer systems. In an S-record, a two-level encoding method is used to transform each byte of binary data into two printable characters; therefore, the transportation process can be visually monitored and the file data can be more easily edited.

When viewed by the user, S-records are essentially character strings made up of several fields, in which pairs of characters are interpreted as hexadecimal values from 1- to 2-byte length, representing a count, an address, a data record, or a checksum. Internally, each record is viewed as a sequence of byte values representing characters. To be compatible with teletype units, S-records may be no longer than 70 bytes. Since 10 bytes are required in each record for the type, byte_count, address, and checksum fields, the variable-length data field may be allocated at maximum 60 bytes. This translates to 60 characters or 30 character pairs or bytes of data per record, from the user viewpoint.

The internal format of an S-record comprises five fields, as shown below:

type	byte_count	address	data	checksum
------	------------	---------	------	----------

where the fields are composed as follows:

Field	Size (bytes)	Contents
type	2	Record type--S0, S1, S2, or S9. The two bytes are hexadecimal, encoded directly from byte values.
byte count	2	The count of the character pairs in the record, excluding the type and checksum fields. The high and low order hexadecimal digits of the actual byte value are individually represented as two hexadecimal bytes in the S-record.

Field	Size (bytes)	Contents
address	4-6	The address at which the data field is to be loaded into memory. The high and low order hexadecimal digits of each actual type value are individually represented as hexadecimal bytes in the S-record.
data	0-60	Memory loadable data or descriptive information. High and low order hexadecimal digits of successive, actual byte values are individually represented as hexadecimal bytes in the S-record.
checksum	2	The least significant byte of the one's complement of the sum of the values represented by the pairs of characters making up the byte count, the address, and the data fields. The high and low order hexadecimal digits of the actual checksum value are individually represented as two hexadecimal bytes in the S-record.

Data blocks output by the linker may contain S-records of the following types:

S0 The header record for each block of data. Subfields in the data field may be:

module name = 20 bytes
version number = 2 bytes
revision number = 2 bytes
description = 0 to 36 bytes

Each of the subfields is composed of bytes, whose associated character, when paired, represent 1-byte hexadecimal values in the case of the version and revision numbers, or represent the encodement of the module name and description specified by the user with the interactive IDENT command.

- S1** A record containing data and the 3-byte address at which the data are to reside.
- S2** A record containing data and the 3-byte address at which the data are to reside.
- S9** A termination record for a block of S-records. The address field may optionally contain the address, specified by the user with the ENTRY command, to which control is to be passed. If not specified, the first entry point specification encountered in the object module input will be used. There is no data field.

9.8 Motorola Pascal Register Usage

This section contains a description of the Pascal register usage and calling conventions and includes descriptions of procedures to be followed when linking more than one procedure or using assembly language files.

An assembly language routine may be called externally by a Pascal program using normal Pascal argument passing. Such a routine may:

- Perform a function not available in Pascal; i.e., data manipulation or I/O not provided in the applicable library, or some mathematics not supported by Pascal.
- Optimize code to be used repetitively in a real-time environment. The Pascal compiler does optimize, but a user-written assembly language routine may be shorter and faster.

Program Preparation

There are two requirements that must be satisfied to include an assembly language subroutine in a Pascal program. First, the external assembly language routine must be declared in the Pascal program. This is done by declaring a level 1 procedure or function (i.e., one contained only by the main program) using the forward directive. A good place for these declarations is prior to the first nonexternal heading.

For example:

```
FUNCTION MSGCOPY(m:Ptrqdata):Ptrqdata;  
FORWARD;
```

The external assembly language subroutine may then be called just as any Pascal procedure or function. The second requirement concerns the file that contains the assembly language routine. This file must have an entry point, that has been declared external with an XDEF, with the same name as the procedure or function in the Pascal program. The assembler must be informed that the subroutine is to be included in section 9. A 'SECTION 9' directive at the beginning of the assembly language subroutine file accomplishes this.

Calling a Routine

Calling an assembly language routine is identical in format (and its run-time requirements are identical in system usage) to a regular function or proce-

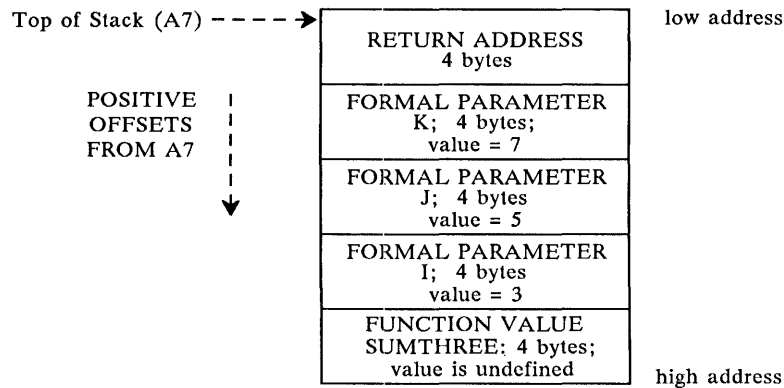
ture call in Pascal. Parameters, for example, are placed on the top of the stack, beneath the return address, in the order they are declared; the first parameter is stacked first and the last parameter is nearest the top of the stack. If the assembly language routine is declared a function, the space for the return value is below the first parameter on the stack (i.e., the address contained in A7 plus a positive displacement). For example, given the declaration and call in the following Pascal program fragment:

```
FUNCTION SUMTHREE(I,J,K:INTEGER):INTEGER; FORWARD;
```

```
BEGIN
```

```
A:= SUMTHREE(3,5,7);
```

the stack would look as follows upon entry to the assembly language subroutine named Sumthree:



The size of parameters depends on the type.

A VAR parameter passes a four-byte address of the actual parameter that can be used to reference the actual parameter via indirection. A value parameter passes the value of the expression that corresponds to the formal parameter.

Boolean parameters occupy two bytes on the stack, but only the byte closer to the top of the stack contains valid data. This byte has the value of one for true and the value of zero for false.

Character parameters use two bytes on the stack, but only the byte closest to the top of the stack contains valid data. This byte has the value of the ASCII code for the character passed in it.

Integer parameters occupy four bytes on the stack. They are stored as 32-bit two's-complement numbers. Integer subrange types that fall into the range -128 to 127, inclusive, use type bytes on the stack, but only the byte closer to the top of the stack contains valid data. They are stored as 8-bit two's-complement numbers. Integer subrange types that extend outside of the range -128 to 127, inclusive, but are within the range -32768 to 32767, inclusive, use two bytes on the stack. They are stored as 16-bit two's-complement numbers.

Real parameters occupy four bytes on the stack, with the sign bit being closest to the top of the stack. Real parameters occupy eight bytes on the stack, with the sign bit being closest to the top of the stack. Xreal parameters occupy ten bytes on the stack, with the sign bit being closest to the top of the stack.

Set parameters require eight bytes on the stack, with the byte nearest the top of the stack containing bits 63-56 and the byte farthest from the top of the stack containing bits 7-0.

Arrays and records occupy a number of bytes equal to their length, plus one if they are of an odd length. The filler byte is the byte farthest from the top of the stack.

Strings should always be passed to assembly language routines as VAR parameters, due to the complexity of determining their actual size on the stack.

Pointers require four bytes on the stack and they contain the address of the variable they reference.

Registers

The assembly language subroutine is responsible for preserving the value of registers A3, A5, and A6 during its execution. It is also responsible for removing from the stack all parameters passed to it by the Pascal program, and for storing a value in the return value location on the stack if the subroutine was declared as a function.

The values of the A5 and A6 registers may be of use to the assembly language routine, since A5 points to the base of the global variable area and

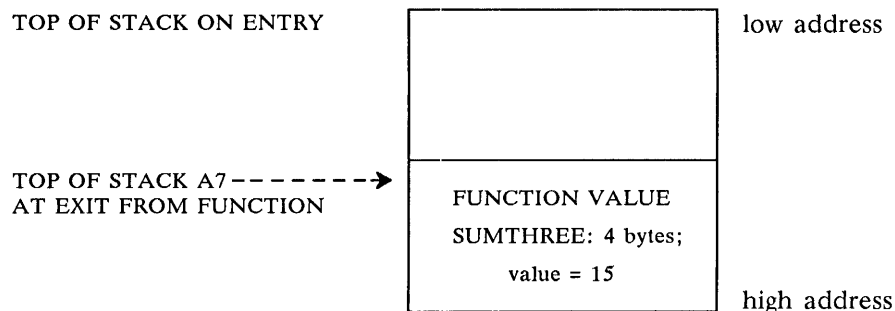
A6 points to the base of the local variable area of the procedure or function that was being executed when the assembly language routine was called. To reference a variable in either of these areas, a negative displacement from the register must be used.

The assembly language subroutine is free to use the space between the top of the stack (pointed to by A7) and the top of the heap for local data storage. The address of the top of the heap is kept in the long word which is located in memory at a positive offset of four from the address in register A5.

If A7 ever contains an address that is less than the address of the top of the heap, a stack/heap overflow condition has occurred. If a stack/heap overflow has occurred, then both the stack and the heap may contain invalid data.

Control may be returned to the Pascal program by means of either a return from subroutine instruction or a jump indirect through an address register which contains the return address. No matter which method is used, it is up to the assembly language subroutine to adjust the stack so as to remove the passed parameters. If the assembly language routine returned a function value, then A7 should point to that location on the stack where the space was reserved for the return value prior to the call. If the assembly language routine did not return a function value, A7 should point just below where the first parameter was pushed on the stack.

The following is a picture of the stack for the SUMTHREE routine, seen earlier, just before the return to the Pascal program:



9.9 Commhead Format

Commhead

RECORD

Actlist: Ptrnamedentity ;	{ Active functions }
Actlock: Lock ;	{ Lock on Actlist }
Mischead: Lock ;	{ Lock on cheader fields }
	{ not otherwise locked }
Fcnkill: Ptrnamedentity ;	{ Dying functions }
Killer: Ptrnamedentity ;	{ Their killer }
Auclock: Lock ;	{ Lock on alpha ↑/ .Usage }
Updlock: Lock ;	{ Lock on Pasuph/t }
Dcr: Ptrdcr ;	{ THE DCR }
RedAmbient: Int16;	{ ambient light base color... }
GreenAmbient: Int16;	{ ... for shading ... }
BlueAmbient: Int16;	{ ... see SHADEINTF.DOC }
Packet_Received: BOOLEAN;	{ for transfer indicator }
NotUsed2: Int8;	
Rdyuph: Ptravupblk ;	{ Head: ACP fmt'd updates }
Rdyupt: Ptravupblk ;	{ Tail: ... }
	{ Above here, known to ACP? }
FSpointer: Integer;	{ makes FS ↑/ .error available V171 }
Dtroy_alpha: Ptralphablk ;	{ Destroy's name }
Dtroy_FI: Ptrnamedentity ;	{ its function inst. }
Dtroy_IS: Ptrfcninputs ;	{ its input set fcn instance blocks }
	{ in priority order }
Prilist: Fcn_pri_array ;	{ head }
Pritail: Fcn_pri_array ;	{ tail }
Prilock: Fcn_lock_array ;	{ Locks on each }
Chfont: Ptrnamedentity ;	{ Standard character font }
G_msglist: Ptrmsglist;	{ Canned messages }
Hashlock: Lock ;	{ Lock on the hash table & lth }
Hashlength: Int16 ;	{ Length of the hash table }
Hashtable: Ptrhash ;	{ Dictionary of all names }
Parsedict: Ptrdictarr;	{ Normal command dictionary }
ParseXcode: PtrXcodearr;	{ expanded Pcodes for Normal }
	{ command syntax }

Head1: INTEGER;	{ really a PtrZHead: PS340 ScanLine- }
Head2: INTEGER;	{ ZBuffer head of edgepair linked-lists }
Functdict: Ptrfdictarr;	{ Dictionary: function names }
Functable: Ptrfnarray;	{ Table: function specs }
PCperGCP: PtrPCpGCP ;	{ PC report stats for each GCP }
FNany: Boolean ;	{ Should we bother to time fns }
FNptrlock: Lock ;	{ Gain right to change fnper* fields. }
Fnperuser: Ptrfnpuser ;	{ Per user fn report }
FnperGCP: PtrfnpGCP ;	{ Per GCP fn report }
Fcn_nact: Ptrintarray ;	{ # of function activations }
Fcn_ttim: Ptrintarray ;	{ total running time of fcns }
Fcn_mtim: Ptrintarray ;	{ maximum running time of fcns }
Last_ci_n: Int16 ;	{ ID number for CI; locked by Hashlock }
Ini_sav_st: Ptrsavstate ;	{ initial save state }
Std_chfont: Ptralphablck ;	{ standard char font }
Password: Ptrqdata;	{ password qstring }
Upsync: int16 ;	{ sync level (lock w/ Updlock) }
Notused4: Integer; Crash_dcr: Ptrdcr;	{ for system wide crashes }
Crash_lod: Ptrnamedentity;	{ for system crash messages }
TwoK_Location: PtrTwoK;	{ Loc to save MM for 2-k acp }
setup_tables: ptrsetup;	{ Terminal setup information }
Plotinprog: Boolean;	{ Plot in progress }
PIProgLock: Lock;	{ Plot in progress lock }
Ibm_table: Ptribmtrn;	
Ibm_device: Ptribmdevice;	
ASC_IBM_conv_table: Ptrcnvtable;	
Mvup0_pad: int8 ;	{ so either here }
Mvup1_pad: int8 ;	{ halt updates iff noupdates }
Noupdates: Boolean ;	{ is false used in wrtback }
Vup1_pad: int8 ;	{ pad Noupdates to .1 }
Vup2_pad: int8 ;	{ to speed value updates }
Vup3_pad: int8 ;	
WhoAllPlot: ARRAY [1..4] OF CHAR;	{ Which user has a plotter }
	{ allocated to self }
WhoPILock: Lock;	{ Plotter allocation lock }
HCPiniComp: Boolean;	{ No error during plot init }
PliniLock: Lock;	{ Plotter initialize lock }
MemOKlock: Lock;	{ Multiple GCP requires }
MemOKavailable: Arrmemok ;	{ bytes left }

MemOKnumallocated: Arrmemok ; { bytes initially allocated }
 User_scopes: ARRAY [0..3] OF Scopearray; { Log-phys scope map per user }
 Schedstuff: Ptrschedarray ; { for experiments }
 Howtorun: pGCPshowtorun ; { CPU allocation }
 tap: Ptrtaprecord ; { For shoulder tap response }
 Finuph: Ptravupblk; { Finished update head }
 Finupt: Ptravupblk; { Finished update tail }
 Notused5: Int16; UpKillFcn: Ptrnamedentity; { Update killer function }
 UpKillFlg: Int8; { Update killer flag }
 Type340: Char; { Type of 340 system }
 Frametime: Int16; { Seconds for frame--ACP timeout }
 Maxframetime: Int16; { Max Seconds for sectioning }
 { frame--ACP timeout }
 GoOn: goontype ; { Keepgoing in mass memory }
 VopFunction: Ptralphabl; { viewing operation function }
 Mcrash: tCrashinfo ; { Info about crash in master }
 Scrash: tCrashinfo ; { Info about crash in slave }
 WrtBackFcn: Ptralphabl; { Write back data function }
 Pasuphs: ARRAY [MemOKindex] OF Ptravupblk; { Pascal update listheads }
 Pasupts: ARRAY [MemOKindex] OF Ptravupblk; { Pascal update listtails }
 Fmtfcns: ARRAY [MemOKindex] OF Ptrnamedentity; { Update formatters }
 mem_thresh: INTEGER; { No big carves if they'll go below this }
 no_mem_on: BOOLEAN; { Nomemsched is running }
 IBM3270_saved_variables: Ptribm3270;

9.10 Operation and Data Node Formats

Operation Nodes

An operation node is a data structure that modifies the state of the display processor. As shown in Figure 9.10-1, an operation node consists of an integer that indicates this display structure is an operation node (=1), an integer that specifies the particular type of operation node, the descendent alpha, and a variable number of fields required by that particular type of operation node. For any operation node, bit 15 of the operation type is a conditional bit. If this bit is set, and if bit 15 (the blink bit) in the Condition Mask of the ACP State is zero, then the associated operation node is not performed. In all other cases, the operation node is performed. In all cases, the descendant of the operation node is traversed. Figures 9.10-2 to 9.10-28 detail each of the operation nodes.

A box shown as

--

 indicates a long word (32-bits).

NOTE

Motorola uses the following Byte ordering:

AO	AO+1
1	2
3	4
AO+2	AO+3

This is different than the Byte ordering on a VAX:

AO+1	AO
2	1
4	3
AO+3	AO+2

Operation Node Formats

Operation Node		1
C	Operation Type	
Descendent Alpha		
Field 1		
Field 2		
⋮		
Field n		

Figure 9.10-1. General Operation Node Format

NOTE

A “C” in the left corner of the Operation Type block indicates the conditional bit (bit 15).

	Operation Node		1	
	C	Operation Type		0 = Viewcon
	Descendent Alpha			
Real # with Implied Exponent of 0	{	X Center		{ Viewport Center
		Y Center		
		Z Center		
		X Size		{ Viewport Size
		Y Size		
		Z Size		

Figure 9.10-2. Viewport

Operation Node		1	
C	Operation Type	2	= Matcon2
Descendent Alpha			
Exponent			
M (1,1)			
M (1,2)			
M (2,1)			
M (2,2)			

Figure 9.10-3. Character Rotate, Character Scale, Character Size, Matrix_2x2

	Operation Node	1	
C	Operation Type	3	= Matcon3
	Descendent Alpha		
	Exponent		
	M (1,1)		
	M (1,2)		
	•		
	•		
	•		
	M (1,2)		
	Tran Flag	0	0 = No Translation Follows

Figure 9.10-4. Rotate, Scale, Matrix_3x3

	Operation Node	1	
C	Operation Type	4	= Matload4
	Descendent Alpha		
	Exponent		(Row 4)
	Exponent		(Row 1-3)
	M (1,1)		
	M (1,2)		
	•		
	•		
	•		
	M (4,4)		

Figure 9.10-5. Window, Eye Back, Field_of_View, Matrix_4x4

	Operation Node	1	
C	Operation Type	5	= Translate
	Descendent Alpha		
	Exponent		
	Tx		
	Ty		
	Tz		

Figure 9.10-6. Translate

	Operation Node	1	
C	Operation Type	6	= IncLOD
	Descendent Alpha		

Figure 9.10-7. Increment Level-of-Detail

	Operation Node	1	
C	Operation Type	7	= DecLOD
	Descendent Alpha		

Figure 9.10-8. Decrement Level-of-Detail

	Operation Node	1	
C	Operation Type	8	= Change Bits
	Descendent Alpha		
	Wordindex		
	Offmask		
	Onmask		

**Figure 9.10-9. Set Level-of-Detail, Set Conditional Bit
Set Displays, Set Character Orientation,
Set Contrast, Set Depth_Clipping,
Set Rate External, Set Blinking,
Set Line_Texture**

NOTES

- Wordindex =
- 0 - LOD value
 - 1 - Conditional bits
 - 2 - Line Generator Mask
 - 3 - Enable/PLS Mask
 - 4 - Line Texture value

The changebits operation node is created by several different PS 390 user commands. Its format is shown above. These PS 390 commands determine the wordindex and also the offmask and onmask. When encountering this node in the structure, the ACP processor locates the correct mask according to the wordindex value. It then modifies the bits in this mask by turning OFF the bits indicated in the Offmask and then turning ON the bits indicated in the Onmask.

Command	Wordindex	Bits used	Offmask	Onmask
Set LOD	0	0-15	16#FFFF	LOD value
Set conditional_bit n on	1	n	XX	XX
Set conditional_bit n off	1	n	XX	0
<i>where XX is the 16 bit word with bit n set</i>				
Set csm on	2	6	16#0040	0
Set csm off	2	6	16\$0040	16\$040
Set displays all on	2	12	16#1000	0
Set displays all off	2	12	16#1000	16#1000
Set plotter on	3	11	16#0800	16#0800
Set plotter off	3	11	16#0800	0
Set picking on	3	15	16#8000	16#8000
Set picking off	3	15	16#8000	0
Set depth_clip on	3	2	16#0004	0
Set depth_clip off	3	2	16#0004	16#0004
Set char world_oriented	3	12,13	16#3900	0
Set char screen_oriented	3	12,13	16#3900	16#2000
Set char screen/or/fixed	3	12,13	16#3900	16#1000
Set Line_texture	4	2-8	16#3FC	Texture
Set Line_texture contin	4	2-8,9	16#3FC	16#0020 OR Texture

NOTE

Set CSM and Set Plotter are not supported on the PS 390.

Operation Node		1	= Bit Test Conditional Mask
C	Operation Type	9	
Descendent Alpha			
C Type			
C Bit			

Figure 9.10-10. IF Conditional_Bit, IF Phase (Bit 15)

NOTES

C Type = 0 Bit off
1 Bit on

Operation Node		1	= Value Test Level of Detail
C	Operation Type	9	
Descendent Alpha			
C Type			
C Val			

Figure 9.10-11. IF Level_of_Detail

NOTES

C Type = 2-Les
3-Eq
4-Leq
5-Gtr
6-Neq
7-Geq

	Operation Node	1	
C	Operation Type	10	= Mat3 Trans
	Descendent Alpha		
	Exponent		
	M (1,1)		
	M (1,2)		
	•		
	•		
	•		
	M (3,3)		
	Tran Flag	1	1 = Translation Follows
	Exponent		
	Tx		
	Ty		
	Tz		

Figure 9.10-12. Look At/From, Matrix_4x3

	Operation Node	1	
C	Operation Type	11	= Trypick
	Descendent Alpha		
Real # with Implied Exponent of 0	Y Max		
	Y Min		
	X Max		
	X Min		

Figure 9.10-13. Set Picking Location

	Operation Node	1	
C	Operation Type	12	= Pickname
	Descendent Alpha		
	Previous Pick Node		
	Alpha Pickname		

Figure 9.10-14. Set Picking Identifier

	Operation Node	1	
C	Operation Type	13	= Set Charfont
	Descendent Alpha		
	Char Font Alpha		

Figure 9.10-15. Character Font

	Operation Node	1	
C	Operation Type	14	= Set Color
	Descendent Alpha		
	Hue	Saturation	

Figure 9.10-16. Set Color

NOTES

Saturation = 4 bits (bits 6..3) All bits set is maximum saturation
Hue = 7 bits (bits 14..8) Clear Hue = \emptyset , all bits set is max hue

	Operation Node	1	
C	Operation Type	15	= Set Blink Mode
	Descendent Alpha		
	Ct		
b	////////////////////		
	M		
	N		

Figure 9.10-17. Set Rate

NOTES

Ct = Value counting down to zero;
b = Blink bit; set initially to 1
Blink on for M
Blink off for N

	Operation Node	1	
C	Operation Type	16	= Set Intensity
	Descendent Alpha		
	Intensity flag		If zero, ignore this node.
Real # with Implied Exponent of 0	Z Center		
	Z Size		

Figure 9.10-18. Set Intensity

NOTES

Center is minimum intensity.

Size is the difference between minimum and maximum.

	Operation Node	1	
C	Operation Type	18	= Xform1
	Descendent Alpha		
	Destination Alpha		

Figure 9.10-19. Xform Matrix

	Operation Node	1	
C	Operation Type	19	= Xform2
	Descendent Alpha		
	Destination Alpha		

Figure 9.10-20. Xform Vector

	Operation Node	1	
C	Operation Type	21	= Wrtdata
	Descendent Alpha		
	Progress flag		
	Data node count		
	Vector count		
	max. data node count		
	Progress flag		
	Progress flag		

Figure 9.10-21. Writeback

	Operation Node	1	
C	Operation Type	23	= Perform Viewing Operation
	Descendent Alpha		
	Alpha to Modeled view		
	Pointer to Linked List		
	X'0001'		Exponent I41 - I44
	X'0001'		Exponent I11 - I34
	X'40000000'	I11	} Identity matrix
	X'00000000'	I12	
	⋮		
	⋮		
	X'40000000'	I44	

Figure 9.10-22. Solid_Rendering, Surface_Rendering

	Operation Node	1	
C	Operation Type	24	= Define Sectioning Plane
	Descendent Alpha		
	X'0001'		Exponent I41 - I44
	X'0001'		Exponent I11 - I34
	X'40000000'	I11	} Identity matrix
	X'00000000'	I12	
	⋮		
	⋮		
	X'40000000'	I44	

Figure 9.10-23. Sectioning Plane

	Operation Node	1	
C	Operation Type	26	= Lightpen
	Descendent Alpha		
	Node on/off		
	Control word		
	Delta Position		Position delta limit
	Sample Counter		Sample count to output
	Use New X,Y		
	Not used		
	New X		User specified cross X center.
	New Y		User specified cross Y center.

Figure 9.10-24. Light Pen

NOTES

The Light Pen is not supported on the PS 390.

Node on/off:

- Bit 6 -- Set if you have not triggered this operation node. (USER.NODE.OFF)
- 5-- Set if the GCP wants the ACP to display only a tracking cross. (GCP.NODE.OFF)

Control word:

- Bit 9 -- Set if screen blast enabled. (BLAST.ON)
- 8 -- Set if a tracking cross enabled. (CROSS.ON)
- 7 -- (Set if the debug mode enabled -- company confidential) (LP.DEBUG)

Use new XY:

- Bit 10 -- Set if you specified (X,Y) coordinate is used to position a tracking cross. (NEWXY.ON)

New X and New Y:

They must be values in the range from X'4000' to X'C000'.
Where X'4000' is assumed to be 1 and X'C000' -1.

	Operation Node	1	
C	Operation Type	28	= Matload2
	Descendent Alpha		
	Exponent		
	M (1,1)		
	M (1,2)		
	M (2,1)		
	M (2,2)		

Figure 9.10-25. Text Size

	Operation Node	1	
C	Operation Type	30	= LoadViewport
	Descendent Alpha		
Real # with Implied Exponent of 0	X Center		Viewport Center
	Y Center		
	Z Center		
	X Size		Viewport Size
	Y Size		
	Z Size		

Figure 9.10-26. Load Viewport

	Operation Node	1	
C	Operation Type	32	= BlinkRate
	Descendent Alpha		
	Rate		

Figure 9.10-27. Set Blink Rate

	Operation Node	1	
C	Operation Type	34	= LoadPickbound
	Descendent Alpha		
Real # with Implied Exponent of 0	Y Max		
	Y Min		
	X Max		
	X Min		

Figure 9.10-28. Load Picking Location

Data Nodes

A data node is the display structure primitive that causes data to be drawn by the ACP. A data node consists of an integer that indicates this display structure is a data node (=2), an 8-bit field that specifies the mode of vectors in the data node, an 8-bit integer that specifies the particular type of data node, a 32-bit integer which points to the next data node of identical data type, an integer (n) that specifies the number of vectors, polygons or characters in the data node, a 16-bit integer that specifies the pick index, and either vector data (including polygons) or character data. Vector data consists of the two- or three-dimensional vectors (preceded by polygon attribute information if polygons). Character data consists of an initial translation, spacing information, and the character string. The general format of a data node is illustrated in Figure 9.10-29. A description of the fields can be found in section 2.2.2.4.

Data Node		2
A	Do_Dots	Data Type
Pointer to Next Data Node		
n		
Pick Index		
*	Line Texture	Traverse Count
*	Color	
	•	
	•	
	•	

Figure 9.10-29. General Data Node Format

NOTE: The data formats in figures 9.10–30 and 9.10–31 are not displayable.

Data Node		2
Do_Dots	Data Type	0 = Vec3f0
Pointer to Next Data Node		
n		
Pick Index		
X1 Y1 Z1		
Exponent 1	Intensity 1	d
X2 Y2 Z2		
Exponent 2	Intensity 2	d
• • •		
Xn Yn Zn		
Exponent n	Intensity n	d

Figure 9.10–30. Vector_List N=n X1,Y1,Z1 X2,Y2,Z2 ... Xn,Yn,Zn
Vector-Normalized (Full Vector) – 3D (Vec3f0)

Data Node		2
Do_Dots	Data Type	1 = Vec2f0
Pointer to Next Data Node		
n		
Pick Index		
X1 Y1		
Exponent 1	Intensity 1	d
X2 Y2		
Exponent 2	Intensity 2	d
• •		
Xn Yn		
Exponent n	Intensity n	d

Figure 9.10–31. Vector_List N=n X1,Y1,-- X2,Y2,-- ... Xn,Yn,--
Vector-Normalized (Full Vector) – 2D (Vec2f0)

PS 390 Block-Normalized Data Node Formats

Note: Fields marked by asterisks are not used nor accessed by normal ASCII and GSR commands. The top bit in the second word of each of these formats (labeled "A") is a flag which, if clear, tells the display structure walker to process these fields. This bit is set by default, and there exists no command to clear it. Advanced user-written functions and programs using the physical read/write facility may however, use these fields and clear that flag.

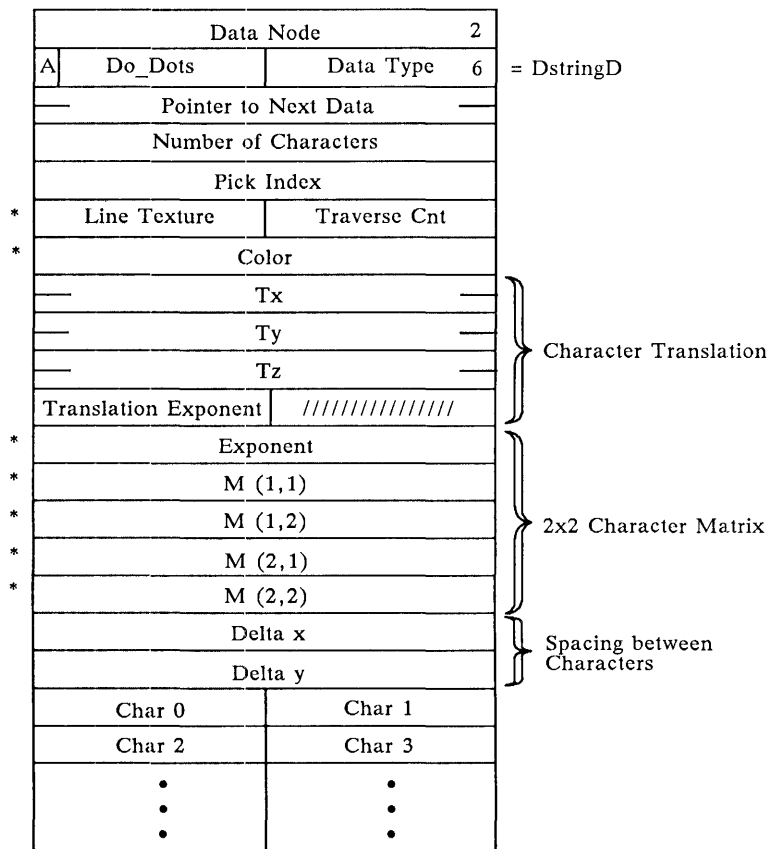


Figure 9.10-32. Characters, Labels Character string (DstringD)

NOTES

Pick Index is always 0 for Characters, 0 for first label, 1 for next label, etc.

A label block with only one label is indistinguishable from a character node.

Data Node			2
A	Do_Dots	Data Type	12 = Vec3bs2
Pointer to Next Data Node			
n			
Pick Index			
*	Line Texture	Traverse Cnt	
*	Color		
	Exponent	Intensity	
	X1		/
	Y1		/
	Z1		d
	X2		/
	Y2		/
	Z2		d
	•		
	•		
	•		
	Xn		/
	Yn		/
	Zn		d

Figure 9.10–33. Vector_List Block N=n X1,Y1,Z1 X2,Y2,Z2 ... Xn,Yn,Zn

Data Node			2
A	Do_Dots	Data Type	13 = Vec2bs2
Pointer to Next Data Node			
n			
Pick Index			
*	Line Texture	Traverse Cnt	
*	Color		
	Exponent	Intensity	
	X1		/
	Y1		d
	X2		/
	Y2		d
	•		
	•		
	•		
	Xn		/
	Yn		d

Figure 9.10–34. Vector_List Block N=n X1,Y1 X2,Y2 ... Xn,Yn

Data Node		2	= Vec2bd0
A	Do_Dots	Data Type 14	
Pointer to Next Data Node			
n			
Pick Index			
*	Line Texture	Traverse Cnt	
*	Color		
	Exponent	Intensity	
0	X1 (H) X1 (L)		
0	Y1 (H) Y1 (L)		
0	Z1 (H) Z1 (L)	/ d	
0	X1 (H) X1 (L)		
0	Y2 (H) Y2 (L)		
0	Z2 (H) Z2 (L)	/ d	
• • •			
0	Xn (H) Xn (L)		
0	Yn (H) Yn (L)		
0	Zn (H) Zn (L)	/ d	

Figure 9.10–35. Vector_List N=n X1,Y1,Z1 X2,Y2,Z2 ... Xn,Yn,Zn

Data Node		2	= Vec2bd0
A	Do_Dots	Data Type 15	
Pointer to Next Data Node			
n			
Pick Index			
*	Line Texture	Traverse Cnt	
*	Color		
Exponent		Intensity	
0	X1 (H) X1 (L)		
0	Y1 (H) Y1 (L)	/ d	
0	X1 (H) X1 (L)		
0	Y2 (H) Y2 (L)	/ d	
• • •			
0	Xn (H) Xn (L)		
0	Yn (H) Yn (L)	/ d	

Figure 9.10–36. Vector_List N=n X1,Y1 X2,Y2 ... Xn,Yn

9.11 Error Types/Error Numbers

There are three crash error types in the PS 390. Each type has a set of error numbers associated with the type. The three types are:

1. System Errors
2. Traps
3. Exceptions

The following is the list of errors for each type.

Type 1 – System Errors

- 1 Track number out of range
- 2 Disk drive not ready
- 3 Disk remains busy after a seek
- 4 Block number out of range
- 6 Lost data during read
- 7 Record not found during read
- 8 Data CRC error during read
- 9 ID CRC error during read
- B Lost data during write
- C Record not found during write
- D Data CRC error during write
- E ID CRC error during write
- F Write fault
- 10 Disk is write protected
- 11 Lost data during format
- 12 Write fault during format
- 14 Disk drive number out of range
- 15 Seek error
- 16 Drive not ready during read
- 17 Drive not ready during write
- 18 Disk not at track 0 after restore command

- 19 Disk busy after restore command
- 1A Track number out of range during format
- 1B Drive not ready during format
- 1C Disk write protected during format
- 1D Time out during read
- 1E Time out during write
- 1F Time out during format
- 64 Wait maybe called with nil argument
- 65 Wait maybe called with a non-function
- 66 Wait maybe, already a function waiting
- 67 Wait maybe, parameter function waiting elsewhere
- 68 Q ship to an unrecognized Namedentity
- 69 Msgcopy, Message type shouldn't be copied
- 6A Msgcopy, Msg type Has structure, unknown to Msgcopy
- 6B Send, 'Me' = nil
- 6C Send, 'Me' not a function instance
- 6D Send, No such output port for this function
- 6E Rem_conn/Add_conn, A1 = nil
- 6F Add_conn, A2 = nil
- 70 Findqueue, Named item = nil
- 71 Findqueue, illegal queue number (queue no. < 0 or queue no. > no. of inputs for function)
- 72 Allinpwait, Nmin > Nmax
- 73 Allinpwait, Nmin < 1
- 74 Tmessage, Waiting and n = 0
- 75 Cmessage, Waiting and n = 0
- 76 Lookmessage, Waiting and n = 0
- 77 Allinputs, Nmin > Nmax
- 78 Allinputs, Nmin < 1
- 79 Fcnotwait, Me = nil
- 7A Findqueue, found a nil queue!
- 7B Waitnextinput, n = 0

7C Anyoutputs, Me = nil
 7D Anyoutputs, illegal outset number
 7E Anyoutputs, no outset where there should be
 7F Fdispatch, function failed to re-queue after running
 80 Text_text, B1 < 0
 81 Char_text, b < 0
 85 Error during disk read
 8D Initial structure not correct
 8E AnnounceUpdate List tail = nil;head < > nil
 8F FormatUpdate Somebody's sleeping in my bed
 90 FormatUpdate Ready Head not nil but Tail is
 91 Bad code file -- illegal Op
 92 ByteIndex Invalid Acpdata type
 93 FormatUpdate, PASCAL Head not nil but Tail is
 94 Vec_size, Invalid Acpdata type
 95 KillUpdate, Updfetch was < 0
 96 KillUpdate, Some one was sleeping in my bed
 97 Vec_bias, Invalid Acpdata type
 99 CntCapacity, Invalid Acpdata type
 9C Unknown brand of Namedentity
 9D Hasstructure knows something I don't
 9E Amuhead not a Qalphapair
 A1 AppendVector, Invalid Acpdata type
 A3 Nomemsched, Bad .Status for a fcn
 A9 Bad update list on ACP time-out
 AA ACP Timeout during initialization
 AB Crashprepare, Name CRASH\$ has not been defined
 AC DecUpdsync, C_header ^ .Updsync < 0
 AD FormatUpdate, Someone waiting in C_header ^ .Updswait already
 AF Someone else waiting in C_header ^ .Killer already
 BO Non-nil Qwait of a dying function
 B3 Microcode won't fit into ACP

B4	Implementation limit on delta waits (2**31)
B8	detected internal inconsistency
B9	detected error (passed a bad parameter)
BA	diskette's parsecode table inconsistent with parser
BD	Bad boundary on binary data xfer
BF	default Devsts contains errors
C0	Inwait, f is already waiting or not a function
C1	Outwait, f is already waiting or not a function
C2	ECO Level of GCP does not support 56K Baud Line
C3	Port 1 Configuration is invalid for 56K Baud Line Support
C9	User generic function stack overflow
CA	Ug_run_cnt has become negative
CB	User generic function has bad alpha (on private queue)
CC	Bad format of MSGLIST .DAT detected
CD	MSGLIST (or code using it) has probably been corrupted
CF	Apparent datastructure incompatibility
D0	Bad MemOKindex detected
D1	routine passed bad parm (e.g., a nil ptr)
D2	Lines to IBM system not active
D3	Floppy disk file INITGPIO.DAT; not found or unable to read
D4	Floppy disk file GPIOCODE.DAT; not found or unable to read
D5	Floppy disk file IBMFONT.DAT; not found or unable to read
D6	Floppy disk file IBMKEYBD.DAT; not found or unable to read
D7	Floppy disk file IBMASCII.DAT; not found or unable to read
D8	IBM GPIO timeout
D9	No. of minimum inputs is negative
DA	No. of maximum inputs < No. of minimum inputs
DB	No. of maximum inputs > # inputs for function
DC	Sendlist detected a bad list
DE	Sendmess: message to be sent is NIL
DF	Caller did not have a lock set already
E0	Curfcn in improper state to call Getinputs

- E1 Cleanin, Curfcn in improper state to call Cleaninp (e.g., have you first called Getinputs?)
- E2 Somebody remembered a forgotten non-fcninstance
- E5 Alpha not already locked by caller
- E6 Confusion in discarding bad message
- E7 Lock not already set by caller
- E8 Probable multiple master GCPs
- E9 RemOne, Curfcn does not have that many inputs
- EA RemOne, Message to be deleted and message pointed to by Curin-puts is not the same
- EB Lock not already set in Gatheraupdate call
- ED Get2locks detected lock already set
- EE Error in semantic routine for polygon vertex
- EF Destination Alpha was not already locked
- F0 Parent not already locked in add/remove from set
- F1 Child not already locked in add to set
- F3 Alpha not already locked in Gpseudoaupdate
- F6 Confusion about locks or decausages
- F7 Unknown tap reason
- F8 Unanticipated state at which to see shoulder tap
- F9 Illegal number of inputs
- FC No existing DCB found for this user
- FD Timeout, Message on input 1 disappeared before fcn could get it
- FE Error while initializing disk drive
- FF Error while reading disk header
- 100 Error while reading disk directory
- 101 THULE.DAT not found on disk
- 102 Error while reading THULE.DAT
- 103 Curfcn was not active at entry
- 104 Viewport not in structure
- 105 Real_simple, number of digits requested out of range ($n < 1$ or $n > 9$)
- 106 Getnextone, illegal queue specified
- 107 Getnextone, msg on head of queue and specified by Curinput do not agree

- 108 Getnextone, no message on queue, but Curinput < > NIL
- 109 ContBlock, nil block
- 10A Timeout when waiting for all on-line GCPs
- 10B Rehash only works first time, only time now.
- 10C No processor has right to issue this tap
- 10D GetVector, Not an Acpdata block
- 10E GetVector, Not a vector Acpdata block
- 10F Invalid qpacket received
- 110 Tolerance on FCnearzero is absurd
- 111 set construct of father has no dummy control block
- 112 function code has to be of type CI to have elements included and removed
- 113 ShadeEnviron node encountered in non PS 340

Type 2 – Traps

- 0 No mass memory on line, or too little to come up
- 1 More OKINTs than NOINTs or > 128 NOINTs
- 2 Free storage block size bad (on request or in free list)
- 3 Attempt to Activate a non-function (or nil) or bad software detected during startup (most commonly, incompatible datastru.sa detected but perhaps invalid startup routine sequencing (if someone has been mucking around with it))
- 4 NEW call failed to find memory, within NOMEMSCHEd
- 5 Attempt to queue where a function is already waiting
- 6 Systemerror(n)
- 7 Badfcode(Fcn)
- 8 Mass Memory Error Interrupt
- 9 Utility Routine not included in this linked system
- A Probable multiple DISPOSE of the same block
- B Block exponent not big enough
- C Attempt to divide with a divisor which is too small in Fix-LongDivide(twice the dividend must be less than the divisor)
- D (Used by Motorola PASCAL)

Type 3 – Exceptions

- 0 Reset: Initial SSP
- 1 Reset: Initial PC
- 2 Bus Error (i.e. attempt to address nonexistent location in memory)
- 3 Address Error (i.e. attempt to access memory incorrectly, for example an instruction not starting on a word boundary).
- 4 Illegal instruction
- 5 Zero Divide
- 6 CHK Instruction
- 7 TRAPV Instruction
- 8 Privilege violation
- 9 Trace
- 10 Line 1010 Emulator
- 11 Line 1111 Emulator
- 24 Spurious interrupt

9.12. F:USERUPD

Updates Function Introduction

Applications for robotics, animation, and simulation require rapidly updated viewing transformations. The Updates Function, F:USERUPD, was created to allow data structures to be updated quickly by transferring data from the host to the PS 390 very rapidly. F:USERUPD is an enhancement included with Graphics Firmware Version A1 that can be used with all PS 390 interfaces.

Transferring data using the F:USERUPD consists of the following steps:

- Instanting the F:USERUPD function. When the function is instanced, it creates 256 names (Function Instance name 001 through Function Instance name 256).
- Creating a display structure using the names created by F:USERUPD, to update the transformation nodes in that display structure.
- Setting up F:USERUPD by sending the bytes that describe SET count, update types, and indices.
- Updating the data structure by sending the data SET.

Note

The Updates Function, F:USERUPD, is preliminary and may contain bugs. The functionality of the F:USERUPD may change in response to feedback from the users of this function. E&S has provided F:USERUPD so that those users who need the ability to perform fast updates via an RS-232 line may use the function to determine if it meets their needs. E&S makes no commitment to maintain the functionality of F:USERUPD in its present form.

Transformation Updates Supported by F:USERUPD

The updates supported by F:USERUPD follow. Each update is identified by a unique number.

<u>Update</u>	<u>Update Types</u>	<u>Number of Bytes Transmitted</u>	<u>Description</u>
1	Rot in X	2	Angle
2	Rot in Y	2	Angle
3	Rot in Z	2	Angle
4	Rot in XYZ	6	Angles for XYZ axis (rotated in order)
5	Tran in X	3	X coordinate
6	Tran in Y	3	Y coordinate
7	Tran in Z	3	Z Coordinate
8	Tran in XYZ	9	XYZ coordinates
9	String	1 string length	string length string
10	Window	18	Xmin, Xmax, Ymin, Ymax,Front, Back
11	Look From	27	From XYZ, At XYZ, Up XYZ
12	Field of View	8	Angle, Front, Back
13	Scale	9	XYZ Scale factors

The angle for rotations and the coordinates for translations have to be sent in a particular format to this function. To minimize the number of bytes transmitted on the communication line between the host and the PS 390, numeric values are represented in angle values or a floating point format.

Angle Values

The angles for the rotations are represented as 65,536ths of 360 degrees. All angles must be positive. The angles are sent as two bytes with the high order Byte being sent first. Representation for some common values follows:

<u>ANGLE</u> (Degrees)	<u>BYTES (HEX)</u> (High) (Low)	
0	00	00
45	20	00
90	40	00
180	80	00
270	C0	00
<u>360*65*,535</u> 65,536	FF	FF

Floating Point Format

Floating-point numbers are represented by three bytes each. The first byte represents a sign bit and a base two excess 64 exponent. The mantissa sign bit is the most significant bit of this first byte. The next 2-byte field is a normalized 17-bit fraction with the redundant most significant fraction bit not represented. The high-order byte of the fraction is sent first. A detailed example is given at the end of this section.

<u>DECIMAL NO.</u>	<u>SIGN/EXPONENT</u>	<u>MANTISSA (HEX)</u>
0	00	00 00
.5	40	00 00
1	41	00 00
-3	C2	80 00
5	43	40 00

Each update will generate an output directed to predetermined data structures which are generated when the function is instanced.

Note

The examples in this document assume that you have some knowledge of how floating point numbers are represented in computers.

Update SET

Updates which are repeatedly carried out are termed as a SET. The number of updates in a SET is the SET COUNT. After the system is initialized, each SET of updates is sent from the host or the keyboard to the PS 390 in the following format.

SET HEADER	Char(6)
Data	Byte
Data	Byte
Data	Byte
Data	Byte
Data	Byte
Data	Byte

Each SET contains a SET HEADER (CHAR(6)) byte to indicate the beginning of data for that SET. This is then followed by data bytes, the number of bytes depending on the types of updates in that SET. For example, if the set contained two updates, a ROT in XYZ and a TRAN in X then the number of data bytes for that SET would be 9 (6 bytes for the three angles and 3 bytes for the X coordinate).

The updates will not be performed until all the data bytes are received for the particular SET of updates. It is your responsibility to ensure that the data bytes are sent in the correct format. Failing to adhere to this will produce unpredictable results or a system crash, as there is no system check for this condition.

The names are generated at the time the function is instanced. For example:

```
MTUP:=F:USERUPD;
```

will create 256 names MTUP001, MTUP002,.....MTUP256.

Initialization SET

Once the function is instanced, data for the types of updates and the index to the associated names have to be sent to the function by the host or from

the keyboard. This information is sent to the function only once. It determines the number of updates in the SET (namely SET COUNT), and (for each update) the number corresponding to the type of update, and the index corresponding to the name this update is to be directed to.

Note

The only way this information can be changed is by re-initializing the function which is accomplished by re-instancing it.

The index is the number corresponding to the name. If the index is 5, then in the previous example, the update will be directed to MTUP005. Once this information is sent to the function by the host or from the keyboard, data can be sent to this function continuously in the format described earlier. The format in which initialization data have to be sent is as follows:

SET HEADER	Char(6)
SET COUNT	Number of Updates in Set
Update Type	Char(1) thru Char(13)
Index	Index into the Names
Update Type	Char(1) thru Char(13)
Index	Index into Names

The name of the function can be any valid PS 390 name. Thus, to update transformation nodes within a data structure, use the following format.

```
World.myupdate := F:USERUPD;
World := BEGIN_STRUCTURE
Myupdate 001 := WINDOW ... ;
Myupdate 002 := ROTATE IN X 0;
END_STRUCTURE;
```

Caution

It is your responsibility to make sure that the numbers corresponding to the Update type and the indices are within limits. No checks are made by the function to ensure that the values are correct.

Failure to do so will result in a system crash without a warning. There are no default values for the window, look from, scale, and field of view commands. All parameters must be sent each time. For example; for the field of view command, the angle as well as the front and back boundaries have to be sent to the function every time when performing a SET of Updates which includes this command.

The function is very easy to use once you are familiar with the PS 390 Command Language. An example demonstrating the use of this function from the keyboard follows. The same outcome can be accomplished by sending the data from the host.

```
PP:=F:USERUPD;  
PPO01:= ROT 0 THEN PPO02;  
PPO02:= TRAN BY 0,0 THEN V;  
V:=VEC 0,0 .5,.5 -.5,.5 0,0;  
DISP PPO01;
```

```
{Set up F:USERUPPD}
```

```
SEND CHAR(6)&CHAR(2)&CHAR(2)&CHAR(1)&CHAR(5)&CHAR(2) TO <1>PP;
```

```
{ Note the format of the data }  
{ Set header, set count, uptype, index, uptype, index }  
{ This indicates a rotate in Y directed to 1 and trans in X }  
{ Directed to 2 }  
{ Update data }
```

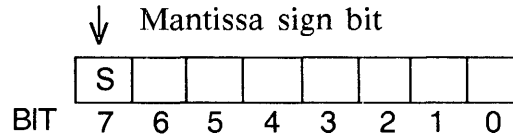
```
SEND CHAR(6)&CHAR(64)&CHAR(0)&CHAR(64)&CHAR(0)&CHAR(0) TO <1>PP;
```

```
{This will rotate the object 90 degrees in Y and translate .5 in X}
```

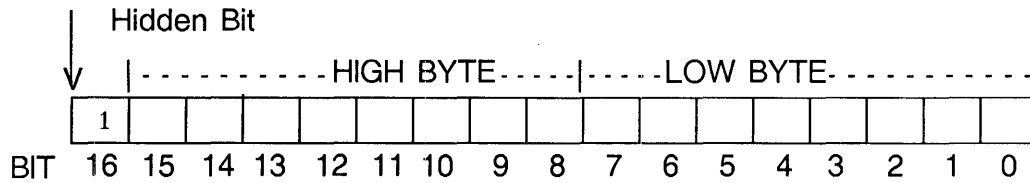
Floating Point Example

EXPONENT BYTE

(Excess - 64)



17 BIT MANTISSA



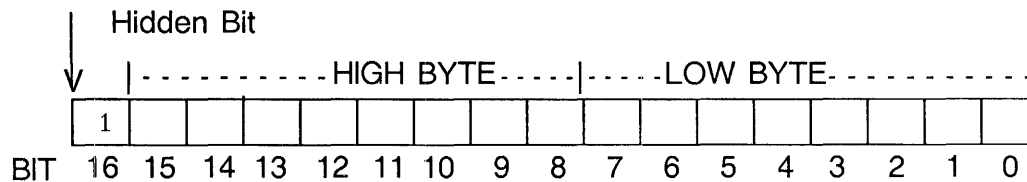
To elaborate on this example, observe the following representation of the floating number 9.0

Real number = $(2^{\text{Exponent}} \times \text{Mantissa}) / (2^{17})$
 [(2^{17}) represents the number of bits in the fractional part.]

Exponent = 68 ($68 - 64 = 4$)

$9.0 = (2^4 \times \text{Mantissa}) / (2^{17})$

Mantissa = 9.0×2^{13}
 $= (2^3 + 1) \times (2^{13})$
 $= 2^{16} + 2^{13}$
 $= \text{Hidden Bit} + \text{Bit 13}$



<u>Number</u>	<u>EXP</u>	<u>HIGH BYTE</u>	<u>LOW BYTE</u>
9.0	44	20	00 (HEX)
9.0	68	32	00 (DECIMAL)

The value of the exponent is the unsigned integer represented by the exponent bits minus the excess of 64. Thus, the exponent 1 is represented as the number 65 (65-64=1) and the exponent -1 is represented by the number 63 (63-64=-1).

By convention, the real number 0 is represented by a value of 0 in the exponent.

The fractional part of the real number is normalized such that $1/2 > [\text{fractional part}] < 1$.

Program Examples

The following programs are contained on the PS 390 magnetic tape labeled PSDIST distributed with the A1 version of the Graphics Firmware. The file UPDATE.DAT also contained on the PS 390 magnetic tape should be loaded on the PS 390 before any of the program examples are run.

PUPDATE.PAS - Pascal Example

PUPDATE.PAS is a program example of how to use the F:USERUPD function to perform fast updates over an RS-232 communications line. This program is contained on the PS 390 magnetic tape labeled PSDIST and uses the E&S supplied GSR routines to send the data to an instance of USERUPD.

The program assumes that the file UPDATE.DAT has been loaded into the PS 390. UPDATE.DAT contains the data structures to be updated, and also instances F:USERUPD, initializes it, and connects it to output <11> of CIRROUTE0. The program updates five rotation nodes, one translation node, and a character string.

The structure being updated is a simple robot arm. Although the program does not demonstrate all updates that are possible using F:USERUPD, it does demonstrate a mechanism for building all of the data types that F:USERUPD acknowledges.

GSRLIB is a VAX logical name for the directory containing the GSR files. The program assumes that the error handling procedure is named ERRHAN and that it resides in the same directory as the GSR files.

```

Program HostUp ( input, output );
CONST
    {GSR const declarations}
    %INCLUDE 'GSRLIB:PROCONST.PAS/nolist'
TYPE
    {GSR type declarations}
    %INCLUDE 'GSRLIB:PROTYPES.PAS/nolist'

    String = Varying [80] of Char;
    Int8 = [byte]0..255;
    Int16 = -32768..32767;

    { a variant record makes it easy to get the pieces of a vax real }
    Vax_real = PACKED RECORD
CASE BOOLEAN OF
    TRUE: ( fr2: 0..127;
junk: Boolean      { can't use this bit }
exponent: 0..255; { place wants sign in exponent }
fr0,fr1: Int8 );
    FALSE: ( r: Real );
        END; {record}

VAR
    current_angle : VARYING [2] OF CHAR; { USERUPD angles are 2 bytes }
    current_real  : VARYING [3] OF CHAR; { USERUPD reals are 3 bytes }
    current_str   : P_VaryingType;  { USERUPD strings are }
        { up to 255 bytes }
    Cnt : Integer;
    Up1,IncUp1 : Real;
    Up2,IncUp2 : Real;
    Up3,IncUp3 : Real;
    Up4,IncUp4 : Real;
    Up5,IncUp5 : Real;
    Up6,IncUp6 : Real;
    PrintString : String;
    Update_set  : P_VaryingType;
    Pr_I : Int16;

    { include the GSR EXTERNAL declarations }
    %INCLUDE 'GSRLIB:PROEXTRN.PAS/nolist'
    { include the GSR error handler }
    %INCLUDE 'GSRLIB:VAXERRHAN.PAS/nolist'

```

```

{ These are the assembler routines to get exponents and mantissas }
PROCEDURE PUPDEXP ( rnum: Real; VAR exp: Int8 ); EXTERN;
PROCEDURE PUPDFRA ( rnum: Real; VAR mhi,mlo: Int8 ); EXTERN;
PROCEDURE Vax_Fp ( rnum: Vax_real; VAR exp,mhi,mlo: Int8 );

{ get the pieces of a USERUPD real from a VAX real }
BEGIN
    PUPDEXP ( rnum.r, exp );
    PUPDFRA ( rnum.r, mhi, mlo );
END;

PROCEDURE R_angle ( angle: Real; VAR ahi,alo: Int8 );

{ Get the pieces of a USERUPD angle from degrees }
CONST
    Factor = 182.0444444; { magic number = 65536/360 }
    {to turn degrees in to 65536's of a circle }
VAR
    itemp: Integer;
    my_angle: real;

BEGIN

{ make any angle its equivalent in the range of 0 to 360 - 1/2**16 }

    my_angle:= angle;
    IF my_angle >= 0 THEN { the angle is positive }
    BEGIN
        REPEAT { make the angle be 0 >= angle < 360 }
            IF my_angle >= 360 THEN
                my_angle:= my_angle - 360;
            UNTIL (my_angle >= 0) AND (my_angle < 360);
        End
    Else { the angle is negative }
    BEGIN
        REPEAT { make the equivalent positive angle }
            IF my_angle < 0 THEN
                my_angle:= my_angle + 360;
            UNTIL (My_angle >= 0) AND (my_angle < 360);
        End;
    itemp:= ROUND(My_angle * factor );
    ahi := itemp DIV 256;
    alo := itemp MOD 256;
END;

PROCEDURE R_real ( r: Real; VAR exp,mhi,mlo: Int8 );

```

```

{ copy a VAX real into the variant record and get the components }
VAR
    rtt: Vax_real;
BEGIN
    rtt.r:=r;
    Vax_fp ( rtt, exp, mhi, mlo );
END;

PROCEDURE P_rot ( angle: Real; VAR Upd_angle: VARYING [len] OF CHAR );

{ make a 2-byte string that is a USERUPD angle }
VAR
    hiangle,loangle: Int8;
BEGIN
    Upd_angle.length:= 2;
    R_angle ( angle, hiangle, loangle );
    Upd_angle[1]:= CHR(hiangle) ;
    Upd_angle[2]:= CHR(loangle) ;
END;

PROCEDURE P_string ( s: String; VAR UPD_string: P_VaryingType );

{ make a USERUPD string; i.e., a 1-byte length and the string }
VAR
    is_i: Int16;
BEGIN
    Upd_string.length:= s.length + 1;
    UPD_string[1]:= CHR(s.length) ;
    FOR is_i := 1 to s.length DO
        UPD_string[is_i +1]:= s[is_i];
    END;

PROCEDURE P_tran ( vec: Real; VAR UPD_trans: VARYING [len] OF CHAR );

{ make a 3-byte string that is a UPD real used for translates }
VAR
    exp,mhi,mlo: Int8;
BEGIN
    Upd_trans.length:= 3;
    R_real ( vec, exp, mhi, mlo );
    UPD_trans[1]:= CHR(exp);
    UPD_trans[2]:= CHR(mhi);
    UPD_trans[3]:= CHR(mlo);
END;

BEGIN

```

```

{ use the GSRs to attach to the PS 390's async interface }

    Pattach('LOGDEVNAM=TT:/PHYDEVTYPE=ASYNC', ERRHAN);

{ Multiplex to CIROUTE<11> the instance of USERUPD is }
{ connected there. }
    PMuxG( 9, ERRHAN );
    Cnt := 3; { go through the sequence 3 times }
    REPEAT
Up1 := 0;
IncUp1 := 1;
Up2 := -90;
IncUp2 := 0.5;
Up3 := 90;
IncUp3 := -0.25;
Up4 := -90;
IncUp4 := 0.5;
Up5 := 0;
IncUp5 := 1;
Up6 := 1;
IncUp6 := -0.0022;
FOR Pr_I := 0 to 720 DO

    BEGIN
Update_set.Length := 0; { initialize the update buffer }
                        { length }
Update_set:= CHR(6);    { every update set must start }
                        { with this character }
                        { get the angle }
P_ROT(Up1,Current_angle);
{ and concatenate it onto the update set }
Update_set:= Update_set + Current_angle;

P_ROT(Up2,Current_angle);
Update_set:= Update_set + Current_angle;

P_ROT(Up3,Current_angle);
Update_set:= Update_set + Current_angle;

P_ROT(Up4,Current_angle);
Update_set:= Update_set + Current_angle;

P_ROT(Up5,Current_angle);
Update_set:= Update_set + Current_angle;

P_TRAN(Up6,Current_real);
Update_set:= Update_set + Current_real;

```

```

PrintString := '      ';
PrintString[3] :=CHR(trunc(Up1) MOD 10 + 48);
PrintString[2] :=CHR(trunc(Up1/10) MOD 10 + 48);
PrintString[1] :=CHR(trunc(Up1/100) MOD 10 + 48);
P_STRING(PrintString,Current_str);
Update_set:= Update_set + Current_str;

{ send the update set to the PS 390 }
PPutG( Update_set, ERRHAN);
{ make sure it goes now }
PPurge( ERRHAN );

{ fix up the angles so that the arm ends up in its }
{ initial position; e.g., up1 goes from 0 to 359 and }
{ back to 0 }
IF Pr_I = 360 THEN
BEGIN
    IncUp1 := -1.0;
    IncUp2 := -0.5;
    IncUp3 := 0.25;
    IncUp4 := -0.5;
    IncUp5 := -1.0;
    IncUp6 := 0.0022;

    END;
Up1 := Up1 + IncUp1;
Up2 := Up2 + IncUp2;
Up3 := Up3 + IncUp3;
Up4 := Up4 + IncUp4;
Up5 := Up5 + IncUp5;
Up6 := Up6 + IncUp6;
    END; { FOR pr_i }
Cnt := Cnt - 1;
    UNTIL Cnt = 0;
    PDetach( ERRHAN );
END.

```

FUPDATE.FOR - FORTRAN Example

FUPDATE.FOR is a program example of how to use the F:USERUPD function to perform fast updates over an RS-232 communications line. This program is contained on the PS 390 magnetic tape labeled PSDIST and uses the E&S supplied GSR routines to send the data to an instance of USERUPD. The program assumes that the file UPDATE.DAT has been

loaded into the PS 390. UPDATE.DAT contains the data structures to be updated, and also instances F:USERUPD, initializes it and connects it to output <11> of CIROUTE0. The program updates five rotation nodes, one translation node, and a character string.

The structure being updated is a simple robot arm. Although the program does not demonstrate all updates that are possible using F:USERUPD it does demonstrate a mechanism for building all of the data types that F:USERUPD acknowledges.

The program assumes that the error handling procedure is named "ERRHND" and that it resides in the same directory as the GSR files.

```

Program Update
REAL Up1,Up2,Up3,Up4,Up5,Up6
REAL IncUp1,IncUp2,IncUp3,IncUp4,IncUp5,IncUp6
CHARACTER Current_angle*2, Current_real*3, Current_str*5
CHARACTER Update_set*20, Printstring*4
C
EXTERNAL ERRHND
C
C
CALL PAtch('LOGDEVNAM=TT:/PHYDEVTYP=ASYNC', ERRHND)
C
C Multiplex to CIROUTE<11>
C
CALL PMuxG( 9, ERRHND )
C
C Do the sequence 3 times
C
  Cnt = 3

1 CONTINUE
  Up1 = 0.
  IncUp1 = 1.
  Up2 = -90.
  IncUp2 = 0.5
  Up3 = 90.
  IncUp3 = -0.25
  Up4 = -90.
  IncUp4 = 0.5
  Up5 = 0.
  IncUp5 = 1.
  Up6 = 1.
  IncUp6 = -0.0022
  DO Pr_I = 0 , 720

```

```

C
C The first character of an update set is always CHAR(6)
C
      Update_set(1:1)= CHAR(6)
C
C Get the angle and put it in the Update set buffer
C
      CALL P_ROT(Up1,Current_angle)
      Update_set(2:3) = Current_angle

      CALL P_ROT(Up2,Current_angle)
      Update_set(4:5)= Current_angle

      CALL P_ROT(Up3,Current_angle)
      Update_set(6:7)= Current_angle

      CALL P_ROT(Up4,Current_angle)
      Update_set(8:9)= Current_angle

      CALL P_ROT(Up5,Current_angle)
      Update_set(10:11)= Current_angle

      CALL P_TRAN(Up6,Current_real)
      Update_set(12:14)= Current_real

      PrintString = '      '
      PrintString(3:3) =CHAR(IMOD(IINT(Up1),10) + 48)
      PrintString(2:2) =CHAR(IMOD(IINT(Up1/10),10) + 48)
      PrintString(1:1) =CHAR(IMOD(IINT(Up1/100),10) + 48)

      CALL P_STRING(PrintString,Current_str)
C
C we know the length of the string so kludge it.
C Fortran thinks the strings length is what it is declared
C to be, Pascal lets you manipulate the length
C
      Update_set(15:19)= Current_str
C
C Send the update set to the PS300
C
      CALL PPutG( Update_set,19,ERRHND)
C
C Make sure it goes now
C
      CALL PPurge( ERRHND )
C
C Fix up the angles so that the arm ends up in its initial

```

```

C position; e.g., Up1 goes from 0 to 359 and back to 0
C
  IF (Pr_I .eq. 360) THEN
    IncUp1 = -1.0
    IncUp2 = -0.5
    IncUp3 =  0.25
    IncUp4 = -0.5
    IncUp5 = -1.0
    IncUp6 =  0.0022

    END IF
    Up1 = Up1 + IncUp1
    Up2 = Up2 + IncUp2
    Up3 = Up3 + IncUp3
    Up4 = Up4 + IncUp4
    Up5 = Up5 + IncUp5
    Up6 = Up6 + IncUp6
    END DO
    Cnt = Cnt - 1
  IF ( Cnt .gt. 0 ) GOTO 1

  CALL PDtach( ERRHND )
END

SUBROUTINE Vax_Fp ( rnum, exp,mhi,mlo)

C
C Redundant routine left in so the FORTRAN looks like the Pascal
C version
C
C PUPDEXP and PUPDFRA are Macro routines to obtain the USERUPD
C exponent and mantissa from a VAX real
C
  REAL rnum
  BYTE exp, mhi,mlo
    CALL PUPDEXP ( rnum, exp )
    CALL PUPDFRA ( rnum, mhi, mlo )
  RETURN
END

SUBROUTINE R_angle ( angle, ahi,alo )

C
C Get the pieces of a USERUPD angle from degrees
C
  REAL Factor, angle, my_angle,temp

```



```

    INTEGER itemp
    BYTE ahi, alo, buff(2)
    EQUIVALENCE ( itemp, buff)
C
C factor is a magic number to turn degrees into
C 65536's of a circle
C
    Factor = 182.0444444  ! = 65536/360

    my_angle= angle
    IF (my_angle .ge. 0.0) THEN ! the angle is positive

1    CONTINUE
        IF (my_angle .ge. 360) THEN
            my_angle= my_angle - 360.0
        END IF
        IF (my_angle .gt. 360) GOTO 1
        ELSE ! the angle is negative
2        CONTINUE ! make the equivalent positive angle
            IF (my_angle .lt. 0) THEN
                my_angle= my_angle + 360.0
            END IF
            IF (My_angle .lt. 0) GOTO 2
        END IF
        Temp = My_angle * factor
        itemp= NINT( Temp )
        ahi = buff(2)
        alo = buff(1)
    RETURN
END

SUBROUTINE R_real ( r, exp,mhi,mlo )

C
C Get the components of a USERUPD real
C
    REAL r
    BYTE exp,mhi,mlo

    CALL Vax_fp ( r, exp, mhi, mlo )
    RETURN
END

SUBROUTINE P_rot ( angle, Upd_angle )
C

```

C Get the components of a USERUPD angle

C

BYTE hiangle,loangle

CHARACTER Upd_angle(*)

REAL angle

CALL R_angle (angle, hiangle, loangle)

Upd_angle(1:1)= CHAR(hiangle)

Upd_angle(2:2)= CHAR(loangle)

RETURN

END

SUBROUTINE P_string (s, UPD_string)

C

C make a USERUPD string; i.e., a 1-byte length and the string

C

INTEGER*2 is_i

CHARACTER s*(*), Upd_string(*)

UPD_string(1:1)= CHAR(LEN(s))

UPD_string(2:) = s

RETURN

END

SUBROUTINE P_tran (vec,UPD_trans)

C

C Make a 3-byte string that is a USERUPD real used for translates

C

REAL vec

CHARACTER Upd_trans(*)

BYTE exp,mhi,mlo

CALL R_real (vec, exp, mhi, mlo)

UPD_trans(1:1)= CHAR(exp)

UPD_trans(2:2)= CHAR(mhi)

UPD_trans(3:3)= CHAR(mlo)

RETURN

END

The following data are contained in the UPDATE.DAT file on the PSDIST magnetic tape and is used with the program examples.

O

CYLINDER:= VEC item N=100

P 1.0000,1., 0.0000 L 0.9686,1., 0.2487

```

L 0.9686,0., 0.2487 L 1.0000,0., 0.0000
P 0.9686,1., 0.2487 L 0.8763,1., 0.4818
L 0.8763,0., 0.4818 L 0.9686,0., 0.2487
P 0.8763,1., 0.4818 L 0.7290,1., 0.6845
L 0.7290,0., 0.6845 L 0.8763,0., 0.4818
P 0.7290,1., 0.6845 L 0.5358,1., 0.8443
L 0.5358,0., 0.8443 L 0.7290,0., 0.6845
P 0.5358,1., 0.8443 L 0.3090,1., 0.9511
L 0.3090,0., 0.9511 L 0.5358,0., 0.8443
P 0.3090,1., 0.9511 L 0.0628,1., 0.9980
L 0.0628,0., 0.9980 L 0.3090,0., 0.9511
P 0.0628,1., 0.9980 L -0.1874,1., 0.9823
L -0.1874,0., 0.9823 L 0.0628,0., 0.9980
P -0.1874,1., 0.9823 L -0.4258,1., 0.9048
L -0.4258,0., 0.9048 L -0.1874,0., 0.9823
P -0.4258,1., 0.9048 L -0.6374,1., 0.7705
L -0.6374,0., 0.7705 L -0.4258,0., 0.9048
P -0.6374,1., 0.7705 L -0.8090,1., 0.5878
L -0.8090,0., 0.5878 L -0.6374,0., 0.7705
P -0.8090,1., 0.5878 L -0.9298,1., 0.3681
L -0.9298,0., 0.3681 L -0.8090,0., 0.5878
P -0.9298,1., 0.3681 L -0.9921,1., 0.1253
L -0.9921,0., 0.1253 L -0.9298,0., 0.3681
P -0.9921,1., 0.1253 L -0.9921,1., -0.1253
L -0.9921,0., -0.1253 L -0.9921,0., 0.1253
P -0.9921,1., -0.1253 L -0.9298,1., -0.3681
L -0.9298,0., -0.3681 L -0.9921,0., -0.1253
P -0.9298,1., -0.3681 L -0.8090,1., -0.5878
L -0.8090,0., -0.5878 L -0.9298,0., -0.3681
P -0.8090,1., -0.5878 L -0.6374,1., -0.7705
L -0.6374,0., -0.7705 L -0.8090,0., -0.5878
P -0.6374,1., -0.7705 L -0.4258,1., -0.9048
L -0.4258,0., -0.9048 L -0.6374,0., -0.7705
P -0.4258,1., -0.9048 L -0.1874,1., -0.9823
L -0.1874,0., -0.9823 L -0.4258,0., -0.9048
P -0.1874,1., -0.9823 L 0.0628,1., -0.9980
L 0.0628,0., -0.9980 L -0.1874,0., -0.9823
P 0.0628,1., -0.9980 L 0.3090,1., -0.9511
L 0.3090,0., -0.9511 L 0.0628,0., -0.9980
P 0.3090,1., -0.9511 L 0.5358,1., -0.8443
L 0.5358,0., -0.8443 L 0.3090,0., -0.9511
P 0.5358,1., -0.8443 L 0.7290,1., -0.6845
L 0.7290,0., -0.6845 L 0.5358,0., -0.8443
P 0.7290,1., -0.6845 L 0.8763,1., -0.4818
L 0.8763,0., -0.4818 L 0.7290,0., -0.6845
P 0.8763,1., -0.4818 L 0.9686,1., -0.2487
L 0.9686,0., -0.2487 L 0.8763,0., -0.4818

```

```

P 0.9686,1.,-0.2487 L 1.0000,1., 0.0000
L 1.0000,0., 0.0000 L 0.9686,0.,-0.2487
;

```

```

{ WORLD SPACE ROTATIONS }

```

```

XMUL := F:MULC;
YMUL := F:MULC;
ZMUL := F:MULC;
XROT := F:XROTATE;
YROT := F:YROTATE;
ZROT := F:ZROTATE;
CMUL := F:CMUL;

```

```

CONN DIALS <1>:<1> XMUL;
CONN DIALS <2>:<1> YMUL;
CONN DIALS <3>:<1> ZMUL;
CONN XMUL <1>:<1> XROT;
CONN YMUL <1>:<1> YROT;
CONN ZMUL <1>:<1> ZROT;
CONN XROT <1>:<2> CMUL;
CONN YROT <1>:<2> CMUL;
CONN ZROT <1>:<2> CMUL;
CONN CMUL <1>:<1> CMUL;

```

```

SEND 150 TO <2> XMUL;
SEND 150 TO <2> YMUL;
SEND 150 TO <2> ZMUL;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <1> CMUL;
SEND 'ROTATE X' TO <1> DLABEL1;
SEND 'ROTATE Y' TO <1> DLABEL2;
SEND 'ROTATE Z' TO <1> DLABEL3;

```

```

SCALE:=F:DSCALE;

```

```

CONN DIALS <4>:<1> SCALE;
CONN SCALE <2>:<3> SCALE;

```

```

SEND 1 TO <2> SCALE;
SEND 1 TO <3> SCALE;
SEND 100 TO <4> SCALE;
SEND 0 TO <5> SCALE;
SEND ' SCALE ' TO <1> DLABEL4;
XVEC:=F:XVECTOR;
YVEC:=F:YVECTOR;
ZVEC:=F:ZVECTOR;
TRAN:=F:ACCUMULATE;

```

```

CONN DIALS <5>:<1> XVEC;
CONN DIALS <6>:<1> YVEC;
CONN DIALS <7>:<1> ZVEC;
CONN XVEC <1>:<1> TRAN;
CONN YVEC <1>:<1> TRAN;
CONN ZVEC <1>:<1> TRAN;

SEND V3D(0,0,0) TO <2> TRAN;
SEND 'TRANS X' TO <1> DLABEL5;
SEND 'TRANS Y' TO <1> DLABEL6;
SEND 'TRANS Z' TO <1> DLABEL7;

UP := F:userupd;
configure a;
disc ciroute0<11>:all;
conn ciroute0<11>:<1>UP1;
finish configuration;
{ initialize the USERUPD function }
send
char(6) { mode character }
&char(7) { do 7 updates }
&char(2)&char(1) { y rot --> UP001 }
&char(1)&char(2) { x rot --> UP002 }
&char(1)&char(3) { x rot --> UP003 }
&char(1)&char(4) { x rot --> UP004 }
&char(2)&char(5) { y rot --> UP005 }
&char(5)&char(6) { x trans --> UP006 }
&char(9)&char(7) { string --> UP007 }
to <1>UP;

INIT DISP;
DISPLAY ROBOT_ARM;

ROBOT_ARM:=BEGIN_S
  { WINDOW }
  Char scale .1 then UP007;
  TR:=TRANSLATE 0,0,0;
  RT:=ROTATE 0;
  SC:=SCALE .2;
  INST FOOTING_COL,UP001;
END_S;
Up007 := Char -1,-.9 'XXXX';

FOOTING_COL:=SET COLOR 0,1 APPLIED TO FOOTING;
FOOTING :=SCALE 3,-2,3 APPLIED TO CYLINDER;

```

```

UP001:=ROTATE Y 0 APPLIED TO BASE_COL;

BASE_COL:=SET COLOR 300,1 APPLIED TO BASE;

BASE:=BEGIN_S
    TRANSLATE -2,7,0      APPLIED TO BASE_L_HUB;
    TRANSLATE 2,7,0       APPLIED TO BASE_R_HUB;
    TRANSLATE -2.5,0,0    APPLIED TO BASE_SUPPORT;
    TRANSLATE 2.5,0,0     APPLIED TO BASE_SUPPORT;
    SCALE 2               APPLIED TO CYLINDER;
    TRANSLATE 0,7,0       APPLIED TO UP002;
END_S;

BASE_L_HUB :=ROTATE Z 90      APPLIED TO CYLINDER;
BASE_R_HUB :=ROTATE Z -90     APPLIED TO CYLINDER;
BASE_SUPPORT:=SCALE .5,8,.5  APPLIED TO CYLINDER;

UP002:=ROTATE X -90 APPLIED TO PRIMARY_COL;

PRIMARY_COL:=SET COLOR 240,1 APPLIED TO PRIMARY;

PRIMARY:=BEGIN_S
    TRANSLATE 2,0,0      APPLIED TO PRIMARY_PIVOT;
    TRANSLATE -.5,13,0   APPLIED TO PRIMARY_L_HUB;
    TRANSLATE .5,13,0    APPLIED TO PRIMARY_R_HUB;
    TRANSLATE -1,-4,0    APPLIED TO PRIMARY_SUPPORT;
    TRANSLATE 1,-4,0     APPLIED TO PRIMARY_SUPPORT;
    TRANSLATE 1.5,-4,0   APPLIED TO PRIMARY_TIE;
    TRANSLATE 0,13,0     APPLIED TO UP003;
END_S;

PRIMARY_PIVOT :=ROTATE Z 90      APPLIED TO PRIMARY_PIVOTC;
PRIMARY_L_HUB :=ROTATE Z 90      APPLIED TO CYLINDER;
PRIMARY_R_HUB :=ROTATE Z -90     APPLIED TO CYLINDER;
PRIMARY_SUPPORT:=SCALE .5,18,.5 APPLIED TO CYLINDER;
PRIMARY_TIE :=ROTATE Z 90      APPLIED TO PRIMARY_TIEC;
PRIMARY_PIVOTC :=SCALE 1,4,1     APPLIED TO CYLINDER;
PRIMARY_TIEC :=SCALE .5,3,.5    APPLIED TO CYLINDER;

UP003:=ROTATE X 90 APPLIED TO SECONDARY_COL;

SECONDARY_COL:=SET COLOR 180,1 APPLIED TO SECONDARY;
SECONDARY:=BEGIN_S
    TRANSLATE .5,11,0 APPLIED TO SECONDARY_HUB;
    TRANSLATE 0,-4,0 APPLIED TO SECONDARY_SUPPORT;
    TRANSLATE .5,0,0 APPLIED TO SECONDARY_HUB;
    TRANSLATE 0,11,0 APPLIED TO UP004;

```

```

END_S;

SECONDARY_HUB      :=ROTATE Z 90      APPLIED TO CYLINDER;
SECONDARY_SUPPORT:=SCALE .5,16,.5 APPLIED TO CYLINDER;

UP004:=ROTATE X -90 APPLIED TO WRIST_COL;

WRIST_COL:=SET COLOR 120,1 APPLIED TO WRIST;

WRIST:=BEGIN_S
  TRANSLATE -.5,0,0  APPLIED TO WRIST_L_HUB;
  TRANSLATE .5,0,0  APPLIED TO WRIST_R_HUB;
  TRANSLATE -.85,-1,0  APPLIED TO WRIST_SUPPORT;
  TRANSLATE .85,-1,0  APPLIED TO WRIST_SUPPORT;
  TRANSLATE 0,2,0  APPLIED TO WRIST_PIVOT;
  TRANSLATE 0,2.5,0 APPLIED TO UP005;
END_S;

WRIST_L_HUB  :=ROTATE Z 90  APPLIED TO WRIST_HUB;
WRIST_R_HUB  :=ROTATE Z -90 APPLIED TO WRIST_HUB;
WRIST_SUPPORT:=SCALE .35,3.0,.35 APPLIED TO CYLINDER;
WRIST_PIVOT  :=SCALE 1.5,-.5,1.5 APPLIED TO CYLINDER;
WRIST_HUB    :=SCALE 1,.5,1      APPLIED TO CYLINDER;

UP005:=ROTATE X 0 APPLIED TO HAND_COL;

HAND_COL:=SET COLOR 60,1 APPLIED TO HAND;

HAND:=BEGIN_S
  SCALE 1.5,-.5,1.5 APPLIED TO CYLINDER;
  ROTATE Y 0  APPLIED TO UP006;
  ROTATE Y 90 APPLIED TO UP006;
  ROTATE Y 180 APPLIED TO UP006;
  ROTATE Y 270 APPLIED TO UP006;
END_S;

UP006:=TRANSLATE 1,0,0 APPLIED TO FINGER;

FINGER:=SCALE .2,2,.2 APPLIED TO CYLINDER;

CONN CMUL  <1>:<1> ROBOT_ARM.RT;
CONN SCALE <1>:<1> ROBOT_ARM.SC;
CONN TRAN  <1>:<1> ROBOT_ARM.TR;
>

```

Host Communication Example

Listed below is a FORTRAN subroutine called SHIP, which can be used in a VAX/VMS environment to provide easy host communication with a PS 390 data structure via the F:USERUPD function. This is a method to buffer USERUPD commands if you do not run the GSRs.

Notice particularly the Configuration Mode statements mentioned in the subroutine's Comments Section. These statements must be included in your data structure to provide a link between the PS 390's host communication mechanisms and the F:USERUPD function.

This subroutine is not contained on the PS 390 magnetic tape labeled PSDIST.

```
C
C ROUTINE TO SHIP A BUFFER IN COUNT MODE TO CIROUTE0<8>.
C
C This routine sends the indicated string to CIROUTE0<8> of the PS 390.
C (The terminal must have been set to modes
C TTSYNC,NOWRAP,NOBROAD,EIGHTBIT.)
C The PS 390 must have been configured with the following code:
C
C CONFIGURE A;
C DISCONNECT CIROUTE0<8>:ALL;
C USERNOP1 := F:NOP;
C CONNECT CIROUTE0<8>:<1>USERNOP1;
C FINISH CONFIGURATION;
C
C FORTRAN calling sequence:
C
C CALL SHIP(JBUF,ICT,IRATE)
C
C Where:
C
C JBUF is the LOGICAL*1 buffer containing the data to be shipped
C (if ICT>0).
C ICT is the INTEGER*2 byte count for JBUF. If ICT=0, then initial-
C ization is assumed, and the first byte of JBUF is taken as the name
C of the USERUPDATES function to be initialized.
C IRATE is INTEGER*2 maximum update rate, in frames per second. If
C this value is zero, update proceeds without any delay. (This
C argument is noticed only when ICT=0.)
C
```



```

      SIZ(2)=JKT(2)
      SIZ(3)=JKT(1)
      T=T+DT
C
30  TIME=SECNDS(TIME0)
      IF(TIME.LT.T) GO TO 30
C
C This system call sends the 4-byte count mode prefix to the PS 390.
C
C
      STAT2=SYS$QIOW(,%VAL(CHAN),%VAL(IO$_WRITEVBLK+IO$_NOFORMAT
1   +IO$_CANCTRLO),IOSB2,,,SIZ,%VAL(4),,,, )
C
C
C This system call sends ICT bytes form buffer JBUF to the PS 390.
C
C
      STATUS=SYS$QIOW(,%VAL(CHAN),%VAL(IO$_WRITEVBLK+IO$_NOFORMAT
1   +IO$_CANCTRLO),IOSB,,,JBUF,%VAL(ICT),,,, )
C
50  IF(STATUS.EQ.1.AND.STAT2.EQ.1.AND.IOSB2(1).EQ.1) RETURN
      TYPE *, 'COMMUNICATION ERROR! - STATUS = ',
1     STATUS,STAT2,IOSB(1),IOSB2(1)
      STOP
      END

```

Floating Point Conversion Routine

Following are two program segments, one written in FORTRAN and the other written in VAX assembly language. These segments convert standard VAX 4-byte floating-point into the floating-point format expected by the F:USERUPD function. Both program segments are contained on the PS 390 magnetic tape labeled PSDIST in the file TUB.MAR.

```

C
C CONVERSION ROUTINE FOR PS 300 FLOATING POINT NUMBERS
C
C Calling sequence:
C
C CALL FPCVT(A,JNUM)
C
C Where:
C
C A is the REAL*4 number to be converted.
C JNUM is the LOGICAL*1 string of three bytes to receive the converted

```

```

C result.
C
SUBROUTINE FPCVT(A,JNUM)
REAL A
LOGICAL*1 JNUM(3)
C
CALL PUPDEXP(A,JNUM(1))
CALL PUPDFRA(A,JNUM(2),JNUM(3))
RETURN
END

```

Assembly Language Conversion Routines

```

.TITLE TLIB
.IDENT /01/

; PUPDEXP--Return EXCESS 64 exponent of a real
; Pascal calling sequence
; PROCEDURE PUPDEXP ( r: Real; VAR exp: INT8 );
;
; FORTRAN calling sequence
; SUBROUTINE PUPDEXP( r, exp )
; Where
; R is REAL*4 number to be converted
; exp is BYTE variable to return the exponent

PUPDEXP::
.WORD ^M<R2, R3>
MOVL @4(AP), R2
ASHL #-7,R2, R3 ; Move into position
BICL2 #^XFFFFFFF00, R3 ; Keep only exponent data
BEQL 1$ ; If zero, leave it alone
SUBL2 #-64, R3 ; Else make it excess 64
1$: BICL2 #^X80, R3 ; Clear sign-bit position
ASHL #-8,R2, R2 ; Go get the sign bit
BICL2 #^XFFFFFF7F, R2 ; Clear all but sign bit
BISB3 R2,R3,@8(AP) ; Set sign
RET

; PUPDFRA--Return hi and low bytes of a real
;
; Pascal Calling sequence
; PROCEDURE PUPDFRA ( r: Real; VAR mhi,mlo: Int8 );
;
; FORTRAN calling sequence
; SUBROUTINE PUPDFRA( r, mhi, mlo )

```

```

; Where
; r is REAL*4 number to be converted
; mhi, and mlo are BYTE variables to return the High and low bytes
; of the mantissa

```

```

PUPDFRA::
    .WORD ^M<R2,R3,R4>
    MOVL @4(AP),R4
    ROTL #1,R4,R4
    BICB3 #^XFFFFFFF00,R4,@8(AP)
    ROTL #8,R4,R4
    BICB3 #^XFFFFFFF00,R4,@12(AP)
    RET
.END

```