November 1984

This manual explains the installation procedure for BASEWAY and specifies the VAX/VMS parameters that should be set or adjusted to optimize performance of BASEWAY. The Release Notes provide additional information about BASEWAY.

BASEWAY

Software Installation Guide/Release Notes

SUPERSESSION/UPDATE INFORMATION: This is a new document for this release.

ERATING SYSTEM AND VERSION: VAX/VMS V3.5

SOFTWARE VERSION: BASEWAY Version 1.0

ORDER NUMBER:

Digital Equipment Corporation
Manufacturing Field Application Center
24730 Crestview Court
Farmington Hills, Michigan 48018

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license. In addition, the following copyright notice must be included:

The following are trademarks of Digital Equipment Corporation:

ALL-IN-1	DATATRIEVE	TRAX
DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10
VAX	VMS	SBI
DECnet	IAS	PDT
FMS		

CONTENTS

Contents

PREFACE

1.0 Manual Objectives

The BASEWAY Software Installation Guide/Release Notes manual shows how to install BASEWAY on VAX/VMS and provides additional information about defining an application and tailoring the product for specific user applications.

2.0 Audience

This manual is intended for those individuals who must set up and maintain the VAX/VMS operating system and BASEWAY software.

3.0 Prerequisites

Readers of this manual should have a solid understanding VAX/VMS operations and administration, and VAX application software. In addition, a knowledge of the specific requirements of the installation site is essential.

4.0 Structure of This Document

This manual is organized as follows:

Chapter 1: Gives an overview of the BASEWAY system.

Chapter 2: Describes the BASEWAY distribution kit, installation prerequisites, and the actual installation procedure.

Chapter 3: Describes the steps involved in defining an application on BASEWAY.

Chapter 4: Contains Release Notes which you should read before installing BASEWAY. This chapter includes information not included elsewhere in the documentation set, changes made late in the development cycle, software errors, and documentation omissions.

Appendix A: Shows a sample installation procedure.

5.0 Associated Documents

Further information on various topics covered in this manual may be found in the following manuals:

- DECnet VMS System Manager's Guide

  (order number AA-H803B-TE)

- SHOP FLOOR GATEWAY Installation Guide/Release Notes

  (order number XX-12355-01)

- BASEWAY System Programmer's Guide

  (order number XX-12346-01)

- BASEWAY User's Manual and Utilities Guide

  (order number XX-12347-01)

- PROGRAMMABLE DEVICE SUPPORT Installation Guide/Release Notes

  (order number 12365-XX)

- VAX ALL-IN-1 Installation Guide/Release Notes

- VAX DATATRIEVE Installation Guide/Release Notes

- VAX FMS Installation Guide and Release Notes

  (order number AA-L321A-TE)

- VAX FMS Utilities Reference Manual

  (order number AA-L320A-TE)

- VAX PL/I Installation Guide

  (order number AA-J179A-TE)

- VAX Software Installation Guide

  (order number AA-M545A-TE)

- VAX/VMS Command Language User's Guide

  (order number AA-D023B-TE)

- VAX/VMS System Manager's Guide

  (order number AA-M547A-TE)

6.0 Disposition of Software Performance Reports (SPRs)

Questions, problems, and enhancements to Digital software should be reported on a Software Performance Report (SPR) form and mailed to the appropriate Digital office. Only one problem should be described concisely on each SPR form. Please include all programs and data in machine-readable form and reference the SPR form number on the materials.

Preface

CHAPTER 1

OVERVIEW OF BASEWAY

1.1 What Is BASEWAY?

BASEWAY is a tool to define, monitor, and control programmable devices and support various user applications. It can work together with Digital's SHOP FLOOR GATEWAY product which provides the actual communications interface to shop floor devices. Up to four gateways per system are permitted. The gateways are PDP-11/24s or PDP-11/44s (run memory-only RSX-11S operating system).

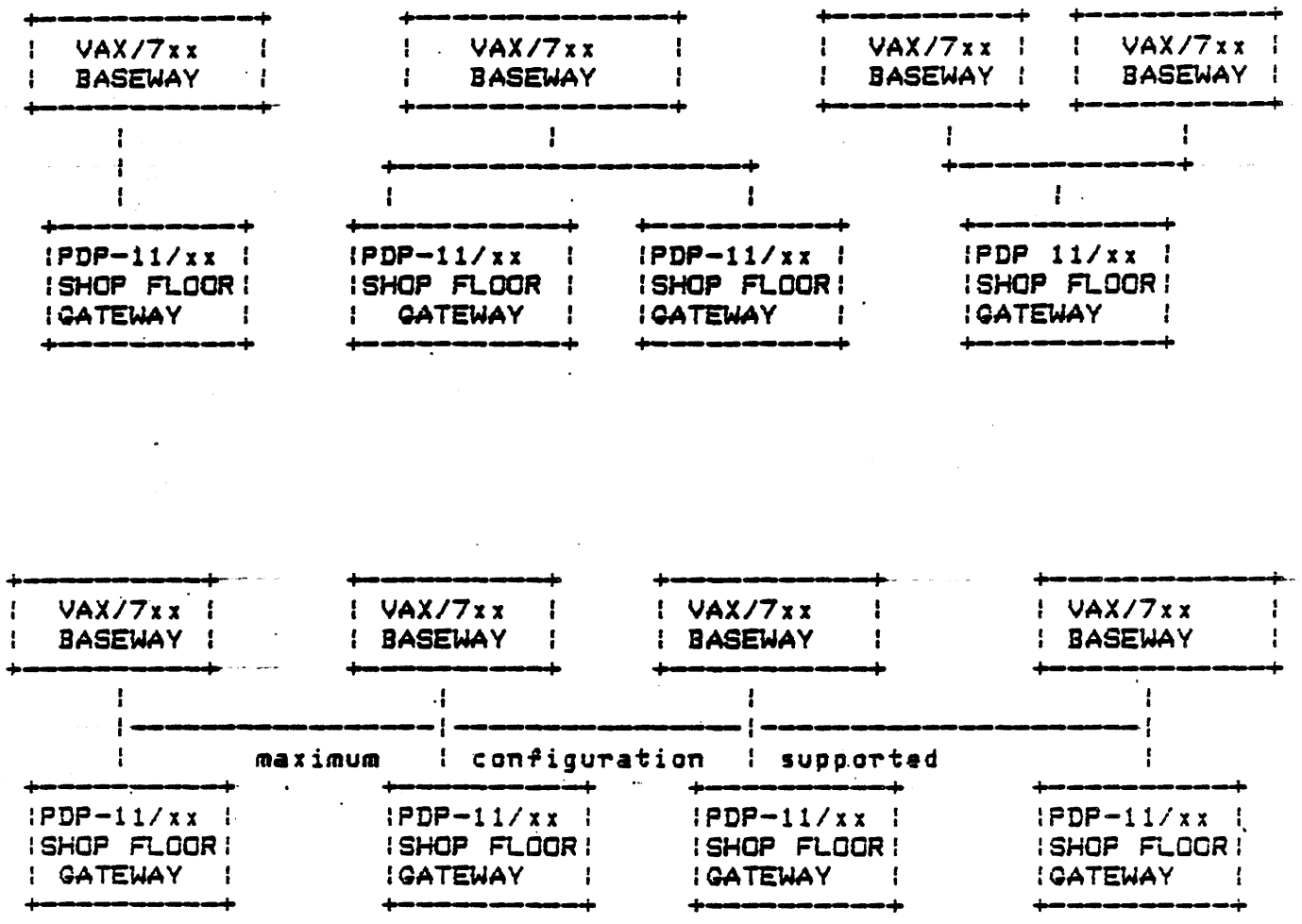


Figure 1. Sample VAX-To-PDP System Configurations

1.2 Hardware Requirements

- VAX 11/750 or 11/780 (memory size dependent upon application)
- PDP-11/24 (or PDP-11/44) 512KB minimum
- Hardware to support DECnet links between the VAX and PDP 11/24, including the automatic boot module for the 11/24.
- Console terminal

1.3 Software Requirements

- VAX DECnet, Version 3.0
- VAX FMS, Version 2.1
- VAX/VMS, Version 3.5

No database for user data or applications other than that necessary to define the network is provided.

In addition, VAX PL/I, Version 2.0, is required if any additional programmable device types are to be added during tailoring.

1.3.1 Optional Software

The following are optional software products, but are suggested for program development:

+ VAX CDD, Version 2.2
+ VAX DATATRIEVE, Version 2.1

- VAX DBMS
- VAX TDMS, Version 1.0
- VAX ACMS, Version 1.0
- VAX ALL-IN-1, Version 1.1

1.4 DATATRIEVE Access

If DATATRIEVE is installed, DATATRIEVE definitions may be added or altered for BASEWAY tailoring. During installation, domain and record definitions are placed in the CDD$TOP.BSL$LIB dictionary. These definitions may be used as desired.

If VAX DATATRIEVE is installed on the target system, the BASEWAY installation procedure will place domain and record definitions for each of the BASEWAY database files in the CDD$TOP.BSL$LIB dictionary. These may be used as necessary to create custom reports of the BASEWAY definitions. However, the format and contents of each of these database files are subject to change in future versions of BASEWAY.

CHAPTER 2

INSTALLING THE SOFTWARE

The system manager should be familiar with the installation process as described herein for use in installing upgrades and updates.

Depending on the mass storage device and the system load, the installation of BASEWAY may take from 15 to 30 minutes.

2.1 Backing Up the System Disk Before Software Installation

It is recommended that the system disk be backed up prior to installation. The procedure for doing the backup is described in the VAX Software Installation Guide.

2.2 Contents of the Distribution Kit

The BASEWAY installation kit is distributed on magnetic tape. All files required to install and tailor the BASEWAY system are contained on the distribution.

Each volume is labeled with an external serial number corresponding to BASEWAY's product number and a unique volume label.

Volume Label	Medium	Contents
BSL010	1600 bpi magnetic tape	BASEWAY installation command procedures and software

NOTE: Be sure to check that the distribution kit you receive contains everything listed in the bill of materials enclosed with it.

2.3 Installation Procedure

2.3.1 Preliminary Requirements

NOTE: BASEWAY requires VAX/VMS Version 3.5 or later.

- o approximately 9000 blocks (BASEWAY peak usage) during installation
- o approximately 8000 blocks (BASEWAY net usage) after installation
- o previous installation of DECnet (v.3.0) and FMS (v.2.1).

2.3.2 Instructions for Installation

Messages are printed at your terminal during the installation procedure. Most are simple "Yes" or "No" questions which require either a Y or N response.

Proceed as follows at the console terminal (user input is shown in uppercase letters):

2.3.2.1 STEP 1: Log In to the System Manager's Account -

1. Log in under a privileged system manager's account, as shown in the following example:

```
CR
Username:  SYSTEM CR
Password:      CR
```

2. Now set up the proper group and user number and set the default directory to SYSUPD as follows:

```
$ SET UIC [1,4]
$ SET DEFAULT SYS$UPDATE
```

2.3.2.2 STEP 2: Invoke VMSINSTAL -

When you invoke VMSINSTAL, it checks the following:

- Are you logged in to the system manager's account?

  It is recommended that you install layered software from the system manager's account. However, any account with the necessary privileges is acceptable.

- Do you have adequate account quotas for installing layered products?

  As long as the quotas listed in Section 2.3.1 are met, you can continue with the installation.

- Is DECnet up and running?

  You should stop DECnet before installing BASEWAY. Although the installation will succeed, problems can occur if someone tries to access any file associated with BASEWAY (including the system HELP files) during the installation.

- Are any users logged in to the system?

  Users should be asked to log out before BASEWAY is installed. Although the installation may succeed, problems can occur if someone tries to use BASEWAY while the installation is in progress.

If any of these conditions are noted, VMSINSTAL will give you an opportunity to stop the installation procedure (see below).

To invoke VMSINSTAL, enter the following:

```
$ @VMSINSTAL BSLnnn ddn:
```

The VMSINSTAL command procedure takes two parameters:

1. product name--for BASEWAY the name always begins with "BSL" and ends with a 3-digit version number. For example, Version 1.6 would be denoted as BSL016. Hereinafter, this document will refer to the version number as Vn.n.

2. device name--device names have the form ddn:, where dd is the device code and n is the unit number. For example, the first floppy diskette drive would be called "DY0:".

NOTE: It is not necessary to use the console drive for installing BASEWAY. However, if you do use it, be sure to replace any media you may have found in the drive when the installation is complete.

```
VAX/VMS Software Product Installation Procedure

It is dd-mmm-19yy at hh:mm
Enter a question mark (?) at any time for help

* Are you satisfied with the backup of your system disk [YES]? Y
```

If you feel that there are conditions which may adversely affect the installation, enter N and the installation will stop. If you wish to continue, enter Y (or press RETURN).

2.3.2.3 STEP 3: Insert the First Installation Kit Volume -

```
Please mount the first volume of the set on ddn:.
```

Insert the first volume of the distribution kit and type "Y" when you are ready to continue.

```
* Are you ready? Y
```

The BASEWAY installation procedure now assumes control. The procedure checks to see that there is adequate disk space to build the product. If not, it issues an error message and terminates. Otherwise, the procedure processes the first volume of the backup save set.

NOTE: If the SYSGEN parameters MOUNTMSG or DISMOUMSG at your site have been set to 1, you will receive a message from OPCOM each time a disk, tape, or floppy diskette is mounted or dismounted. These messages are normally disabled, but if they have been activated and you are installing from a console terminal, they will appear from 1 to 30 seconds after each mount or dismount.

```
%MOUNT-I-MOUNTED, BSLnnn mounted on ddn:

The following products will be installed:

BSL Vn.n

    Beginning installation of BSL Vn.n at hh:mm
```

```
%VMSINSTAL-I-RESTORE, Restoring product saveset A...

Previous logical name assignment replaced
```

At this point, if you need to exit from the installation procedure, you must press CTRL/Y.

NOTE: If you press CTRL/Y, the installation procedure deletes all files it has created up to that point and exits.

```
BSLSTART.COM, the startup command procedure, is used to set up the
environment for the BASEWAY Application Bus.  During installation,
it will be placed in the [SYSMGR] directory of the system root on
which this installation is being performed.  SYS$MANAGER:SYSTARTUP.COM
your system startup procedure, should be modified to invoke this
procedure when the system boots.  However, it will not be necessary
to reboot the system after the installation, since this procedure
is invoked as part of the installation.

%VMSINSTAL-I-MOVEFILES, Files will now be moved to their target
directories...
```

Next, an Installation Verification Procedure (IVP) runs tests to check that the installation procedure was successful.

```
Installation Verification Procedure (IVP) starting

The installation verification of BASEWAY Application Bus vn.n was
successful.


Successful installation of BSL Vn.n at hh:mm
```

BASEWAY images and libraries are now successfully installed. You may now install more products, or you can end the installation procedure. To end the installation procedure, type "EXIT" (or press RETURN).

```
Enter the products to be installed from the next distribution
volume set.
* Products [EXIT]: EXIT

    VMSINSTAL procedure done at hh:mm

$
```

If you removed any media from the console drive before beginning, you can replace it now.

WARNING: VMSINSTAL deletes or changes entries in the logical name tables during the installation. Therefore, if you are going to continue using the system manager's account, you should log out and log in again to restore those tables.

2.3.3 Error Conditions

If the installation procedure or IVP fail for any reason, the following messages are displayed:

```
%VMSINSTAL-E-BSLFAIL, The installation of BSL Vn.n has failed.
%VMSINSTAL-E-IVPFAIL, The IVP for BSL Vn.n has failed.
```

An error during the installation can be caused by one or more of the following conditions:

- insufficient disk space to complete the installation
- insufficient system virtual page count parameter
- insufficient process paging file quota
- insufficient process working set quota
- insufficient system maximum working set
- insufficient system global pages

When you are notified that one of these conditions exists, you should take the appropriate action as described in the message.

To change a system parameter, or to increase an authorized quota value, you may need to contact your installation system manager.

2.4 Backup Procedure After Installation

After installing BASEWAY you should back up the system disk and save the original for future reference. See the VAX Software Installation Guide for information on the proper procedure.

2.5 BASEWAY Directory Structures

The installation procedure creates VAX directory structures on the system disk. The following shows these structures:

Root Directory	SYS$SYSROOT:[BSL]
ALL-IN-1 menu forms	SYS$SYSROOT:[BSL.A1MENU]
Standard menu forms	SYS$SYSROOT:[BSL.MENU]
Source directories	SYS$SYSROOT:[BSL.SOURCE]
Data Processor source files	SYS$SYSROOT:[BSL.SOURCE.DATAPROC]
Source files necessary for adding new programmable device type	SYS$SYSROOT:[BSL.SOURCE.LIBRARY]
BASEWAY system images	SYS$SYSROOT:[BSL.SYSTEM]

2.5.1 Application Directory Structure

Root directory (user-specified at creation)	[xxxxx]
BASEWAY data files	[.DATA]

. (others may be created to suit application needs)

2.6 Systemwide BASEWAY Logical Names

BSL$A1MENU	SYS$SYSROOT:[BSL.A1MENU]
BSL$DEVICE_FILE	BSL$SYSDATA:DEVDEF.DAT
BSL$ENTITY_FILE	BSL$SYSDATA:ENTDEF.DAT
BSL$FORMS	SYS$SYSROOT:[BSL.SYSTEM]
BSL$HISTORY_FILE	BSL$DATA:HISTORY.DAT
BSL$MENU	SYS$SYSROOT:[BSL.MENU]
BSL$POLLING_FILE	BSL$SYSDATA:POLDEF.DAT
BSL$REGISTER_FILE	BSL$SYSDATA:REGDEF.DAT
BSL$SYSDATA	SYS$SYSROOT:[BSL.SYSTEM]
BSL$SYSTEM	SYS$SYSROOT:[BSL.SYSTEM]
BSL$SYSTEM_FILE	BSL$SYSDATA:SYSDEF.DAT
BSL$TERMINAL_FILE	BSL$SYSDATA:TERMDEF.DAT
BSL$USER_FILE	BSL$SYSDATA:USERDEF.DAT

2.7 Groupwide BASEWAY Logical Name

"BSL$DATA" is a groupwide BASEWAY logical name which is created by EVENT_PROC when it is initiated. Logical names that are based on this (such as BSL$HISTORY_FILE) and which are created by the command file SYS$MANAGER:BSLSTART.COM, may not be referenced until an application is running.

Other group logical names which are specific to your application may be created by editing the EVENT_PROC startup command file.

2.8 Complete List of Installed Files

Filename	Purpose
SYS$SYSROOT:[SYSMGR]	
BSLSTART.COM	BASEWAY startup command file
SYS$SYSROOT:[SYSLIB]	
BSLDEF.BAS	VAX BASIC definitions for BASEWAY routines
BSLDEF.FOR	VAX FORTRAN definitions for BASEWAY routines
BSLDEF.H	VAX C definitions for BASEWAY routines
BSLDEF.LIB	VAX COBOL definitions for BASEWAY routines
BSLDEF.PAS	VAX PASCAL definitions for BASEWAY routines
BSLDEF.PLI	VAX PLI definitions for BASEWAY routines
BSLDEF.REQ	VAX BLISS-32 definitions for BASEWAY routines
BSLLIB.OLB	Object library containing BASEWAY routines
BSLPLI.TLB	Text library containing all BASEWAY definitions
BSLSHR.EXE	Sharable image containing BASEWAY routines
BSLSHRBLD.COM	Command file to relink BSLSHR.EXE sharable image
SYS$SYSROOT:[SYSMSG]	
BSLMSG.EXE	Image file containing BASEWAY messages
SFGMSG.EXE	Image file containing SHOP FLOOR GATEWAY messages
SYS$SYSROOT:[BSL.A1MENU]	
A1MENU.FLB	Standard form library for ALL-IN-1 menu support
EDTADDR.COM	Command file to run EDTADDR image from ALL-IN-1
EDTAPPL.COM	Command file to run EDTAPPL image from ALL-IN-1
EDTDEVICE.COM	Command file to run EDTDEVICE image from ALL-IN-1
EDTGATE.COM	Command file to run EDTGATE image from ALL-IN-1
EDTSET.COM	Command file to run EDTSET image from ALL-IN-1
EDTTERM.COM	Command file to run EDTTERM image from ALL-IN-1

EDTUSER.COM	Command file to run EDTUSER image from ALL-IN-1
UTLAPPSTS.COM	Command file to run UTLAPPSTS image from ALL-IN-1
UTLDISDEV.COM	Command file to run UTLDISDEV image from ALL-IN-1
UTLDISPNT.COM	Command file to run UTLDISPNT image from ALL-IN-1
UTLDISTRM.COM	Command file to run UTLDISTRM image from ALL-IN-1
UTLDISUSR.COM	Command file to run UTLDISUSR image from ALL-IN-1
UTLEVTHIS.COM	Command file to run UTLEVTHIS image from ALL-IN-1
UTLGATSTS.COM	Command file to run UTLGATSTS image from ALL-IN-1
UTLNOTYET.COM	Command file to run UTLNOTYET image from ALL-IN-1

SYS$SYSROOT:[BSL.MENU]

MENU.FLB	Standard form library for BASEWAY menu driver

SYS$SYSROOT:[BSL.SOURCE.DATAPROC]

DATAPROC.BLD	Command file to recompile and link DATAPROC
DATAPROC.PLI	Sample polled data sink program

SYS$SYSROOT:[BSL.SOURCE.LIBRARY]

BSLMSG.MSG	Source file for BASEWAY messages
PARSEADDR.MAR	Source file for parse address state tables
PDAPCOMP.PLI	Source file for compile address routine
PDAPDNLOD.PLI	Source file for download device routine
PDAPERROR.PLI	Source file for translate gateway error routine
PDAPGDEF.PLI	Source file for get device default routine
PDAPGMFR.PLI	Source file for get manufacturer routine
PDAPGMFRL.PLI	Source file for get manufacturer list routine
PDAPGMOD.PLI	Source file for get model routine
PDAPGMODL.PLI	Source file for get model list routine
PDAPGNET.PLI	Source file for get network routine
PDAPNXTAD.PLI	Source file for get next address routine
PDAPSTART.PLI	Source file for start device routine
PDAPSTOP.PLI	Source file for stop device routine
PDAPTMFR.PLI	Source file for translate manufacturer routine
PDAPTMOD.PLI	Source file for translate model routine
PDAPTNET.PLI	Source file for translate net routine
PDAPTRAN.PLI	Source file for translate device address routine
PDAPUPLOD.PLI	Source file for upload device routine
SFGMSG.MSG	Source file for SHOP FLOOR GATEWAY messages

SYS$SYSROOT:[BSL.SYSTEM]

DATAPROC.COM	Startup file for sample DATAPROC (called by EVENTPROC
DATAPROC.EXE	Sample polled data sink image
DEFINEAPP.COM	Define application command file
DEVDEF.DAT	Device definition data file
EDITOR.FLB	Form library for BASEWAY editors
EDTADDR.EXE	Device address editor image
EDTAPPL.EXE	Application editor image

EDTAPPMNT.EXE	Application definition utility image
EDTDEVICE.EXE	Device editor image
EDTGATE.EXE	Gateway editor image
EDTSET.EXE	Device set editor image
EDTTERM.EXE	Terminal editor image
EDTUSER.EXE	User editor image
ENTDEF.DAT	Point and group definition data file
EVENTLOG.COM	Startup file for EVENTLOG (called by EVENTPROC)
EVENTLOG.EXE	Event Logger image
EVENTPROC.EXE	Event Processor image
EVTHIS.FDL	File Description Language for system audit file
GATEINIT.COM	Startup file for GATEINIT (called by EVENTPROC)
GATEINIT.EXE	Gateway Initializer image
LOOPTEST.EXE	Loopback test utility image
MENU.EXE	Menu driver image
MODLDC384.EXE	Modicon 184/384 Load/Dump/Compare image
MODLDC484.EXE	Modicon 484 Load/Dump/Compare image
MODLDC584.EXE	Modicon 584 Load/Dump/Compare image
NI.COM	Startup file for NI (called by EVENTPROC)
NI.EXE	Network Interface image
PHYLO.EXE	Allen-Bradley PLC-3 physical to logical converter image
PHYLOGMAN.EXE	Baseway driver image for PHYLO
POLDEF.DAT	Polling set definition data file
REGDEF.DAT	Device register definition data file
SYSDEF.DAT	Application, gateway, and device set data file
TERMDEF.DAT	Terminal definition data file
USERDEF.DAT	User definition data file
UTILITY.FLB	Form library for BASEWAY utilities
UTLAPPSTP.EXE	Application shutdown utility image
UTLAPPSTS.EXE	Application status display image
UTLDISDEV.EXE	Device display image
UTLDISPNT.EXE	Point display image
UTLDISTRM.EXE	Terminal display image
UTLDISUSR.EXE	User display image
UTLEVTHIS.EXE	System audit display image
UTLGATSTS.EXE	Gateway status display image
UTLINIVDR.EXE	Reinitialize gateway database utility image
UTLNOTYET.EXE	Missing function display image
UTLVDRDMP.EXE	Gateway database display utility image

CHAPTER 3

SETTING UP USER APPLICATIONS

After your BASEWAY system has been installed, you may define your user applications.

3.1 Defining Applications

Figure 2 below illustrates some of the steps involved in defining applications. The various steps are described in this chapter.

1 Design and code your application.

2 Run DEFINEAPP.COM to create a user account, login command file, and application.

3 Edit STARTUP.COM to include your application.

4 Change MENU to reflect your application.

Figure 2. Defining An Application

3.1.1 Invoking DEFINEAPP.COM

To create an application, you must invoke the command file DEFINEAPP.COM. This file defines the parameters needed to get a new application running on the system.

DECnet must be up before you invoke DEFINEAPP and start your application.

For information about defining accounts, see Section 3.2.

The following is a sample definition of Application Number "0003". It is a continuation of the installation dialog which started in Section 2.3.2.

CAUTION: Make every effort to type accurately when responding to the prompts in this command procedure.

```
$  SET DEF BSL$SYSTEM
$  @DEFINEAPP

** BASEWAY Application Configuration Procedure **

This procedure will define the parameters needed to get a new
application running on this system.

You may respond to any question or prompt with a "?" if help
is needed.

* Application name ?  FIELD_TEST

* CONFIRMATION:  Is "FIELD_TEST" the correct name ? YES

Creating application FIELD_TEST...

Application number 0003 created...

* Group number for application [11-377] ? 100

* Disk for data directory (if not __DRA0:) ?  cr

* Root directory (if not APP0003) ?  TESTAPP

Creating directory __DRA0:[TESTAPP]...

Creating directory __DRA0:[TESTAPP.DATA]...

Creating Database files...
```

```
Creating BSL$SYSTEM:APP0003.COM...

Creating __DRA0:[TESTAPP]STARTUP.COM...

- Application FIELD_TEST configured.

You should invoke the procedure BSL$SYSTEM:APP0003.COM
to start this new application.

$
```

3.1.2 Starting Up the Application

To start your new application, you must invoke the following procedure:

```
$  @BSL$SYSTEM:APP0003
%RUN-S-PROC_ID, identification of created process is 0007003A
```

You can verify that the application started by looking for the console message, "BASEWAY is Starting."

3.2 Defining Accounts

To define accounts, the following commands should be used:

3.2.1 Manager Account

```
$ SET DEF SYS$SYSTEM
$ RUN SYS$SYSTEM:AUTHORIZE
    ADD APPMGR/OWNER="BASEWAY MANAGER" -
        /UIC=[377,377] -
        /DEVICE=DDCU: -
        /DIRECTORY=[TESTAPP.MGR] -
        /PRCLM=5 -
        /ASTLM=10 -
        /BYTLM=38000 -
        /BIOLM=10 -
        /DIOLM=10 -
        /FILLM=60 -
        /PRIV=(DETACH,TMPMBX,NETMBX,SYSPRV)
```

3.2.2 User Account

```
$ SET DEF SYS$SYSTEM
$ RUN SYS$SYSTEM:AUTHORIZE
    ADD APPUSR/OWNER="BASEWAY USER"/UIC=[377,377] -
        /DEVICE=DDCU: -
        /DIRECTORY=[TESTAPP.USER] -
        /FLAGS=(DISCTLY,DEFCLI,CAPTIVE,LOCKPWD) -
        /PRCLM=5 -
        /ASTLM=10 -
        /BYTLM=38000 -
        /BIOLM=10 -
        /DIOLM=10 -
        /FILLM=60 -
        /PRIV=(NETMBX,TMPMBX)
```

3.2.3 Login Command File

To facilitate menu usage, the manager should set up a login command file for users.

Note the flags, "DISCTLY DEFCLI CAPTIVE", shown in the example used in Section 3.2.2. These flags insure that the user cannot log into this account without executing the login command file. This login command file should invoke the BASEWAY MENU program, and should log out when MENU exits.

The following may be used:

```
$!
$! Login command file
$!
$ IF F$MODE() .EQS. "BATCH" THEN $EXIT
$ SET TERM/INQ
$ SET TERM/NOBRO/FORM/NOWRAP
$ ASSIGN/USER 'F$LOGICAL("TT")' SYS$INPUT
$ ASSIGN/USER 'F$LOGICAL("TT")' TT
$ RUN BSL$SYSTEM:MENU
$ LOG/BRIEF
```

The "SET TERM" commands will set up the user's terminal to support VAX FMS, and will support any local printer that might be connected. The "ASSIGN/USER" commands allow the MENU program to be run from a command file.

3.2.4 Example Accounts

```
UAF>SHOW APPUSR

Username:   APPUSR        Owner:   BASEWAY USER
Account:    BASEWAY       UIC:     [100,200]
CLI:        DCL           LGICMD:  SYS$MANAGER:SYLOGIN.COM
Default Device: DRA1:
Default Directory: [APPUSR]
Login Flags: DISCTLY DEFCLI LOCKPWD CAPTIVE
Primary days:    Mon Tue Wed Thu Fri
Secondary days:                      Sat Sun
No hourly restrictions
PRIO:      4     BYTLM:        38000    BIOLM:          10
PRCLM:     5     PBYTLM:           0    DIOLM:          10
ASTLM:    10     WSDEFAULT:      150    FILLM:          60
ENQLM:    60     WSQUOTA:        200    SHRFILLM:        0
TQELM:    10     WSEXTENT:      1000    CPU:      no limit
MAXJOBS:   0     MAXACCTJOBS:      0    PGFLQUOTA:  10000
Privileges:
  TMPMBX NETMBX


UAF>SHOW APPMGR

Username:   APPMGR        Owner:   BASEWAY MANAGER
  ount:     BASEWAY       UIC:     [100,200]
_I:         DCL           LGICMD:  SYS$MANAGER:SYLOGIN.COM
Default Device: DRA1:
Default Directory: [APPMGR]
Login Flags:
Primary days:  Mon Tue Wed Thu Fri
Secondary days:                      Sat Sun
No hourly restrictions
PRIO:      4     BYTLM:        38000    BIOLM:          10
PRCLM:     5     PBYTLM:           0    DIOLM:          10
ASTLM:    10     WSDEFAULT:      150    FILLM:          60
ENQLM:    60     WSQUOTA:        200    SHRFILLM:        0
TQELM:    10     WSEXTENT:      1000    CPU:      no limit
MAXJOBS:   0     MAXACCTJOBS:      0    PGFLQUOTA:  10000
Privileges:
  DETACH TMPMBX NETMBX SYSPRV
```

3.3 Startup Parameter File

During the startup process, a "Startup Parameter File" is used to control the startup functions of the BASEWAY system. This allows new logical names and subprocesses to be added without modifying portions of the code itself.

The commands in the startup command file are organized into several groups, or sets. The order in which these sets are included is fixed since such things as logical names must be defined before subprocesses are created, etc.

The following example shows how commands are processed by the Event Processor in its initialization phase:

```
BASEWAY startup parameters

SET APPLICATION ID TO 0003
SET LOGICAL NAME BSL$DATA TO DRA0:[APP0003.DATA]
CREATE MAILBOX NAME BSL$MBX_PORT_010
CREATE BASEWAY GLOBAL SECTION
DCL @BSL$SYSTEM:EVENTLOG.COM
DCL @BSL$SYSTEM:NI.COM
DCL @BSL$SYSTEM:GATEINIT.COM
;
;
;-------------------------------------------------
; Insert application-specific logical names HERE
;-------------------------------------------------
;
;
;
;-------------------------------------------------
; Set application-specific flags HERE
;-------------------------------------------------
;
;
;
;-------------------------------------------------
; Run application-specific images HERE
;-------------------------------------------------

DCL @BSL$SYSTEM:DATAPROC.COM
;
```

3.4 Modifying the Menu Facility

BASEWAY allows the menu to be changed to reflect your application. You may wish to modify individual screens or menu flow within the screens.

Two versions of menu support are provided with the system: ALL-IN-1 and MENU. Both are driven by FMS screens. To change the menus, you can edit the forms found in the MENU form library in either the BSL$A1MENU or BSL$MENU directories. Refer to the FMS User's Reference Manual for more detailed information regarding editing of forms.

WARNING: The menu forms released as part of the initial distribution of BASEWAY are subject to change in subsequent releases. Thus, any modifications made to these forms are liable to be lost.

For this reason, it is suggested that you create your tailored menu in the application directory that was created with DEFINEAPP.COM.

3.4.1 ALL-IN-1 Menu Support

The user may want to incorporate the menu structure into his or her own version of ALL-IN-1. The users of the BASEWAY system must be included in the ALL-IN-1 user database and these users must have the BSL$A1MENU:MENU form library defined in their profiles.

3.4.2 BASEWAY Menu Support

The MENU program is distributed for systems without ALL-IN-1 support. It is invoked by each user, and references forms in the BSL$MENU:MENU form library.

3.4.3 Modifying Keyword Options and Their Functions

To modify a keyword option, find the keyword definition screen which contains the keyword you wish to modify. Modify the named data item to do the appropriate function:

Named Data Commands

	MENU	ALL-IN-1
Display Screen	MENU$FORM=frmnam	FORM frmnam
Run Program	MENU$IMAGE=filespec or MENU$COMMAND=filespec	(no equivalent) COMMAND filespec
Leave Program	MENU$EXIT	EXIT

The MENU flow is driven from the contents of the Named Data area of each menu screen. The named data area must use the following syntax conventions:

```
MENU$FORM=<form> [,MENU$LIB=<filespec>]
MENU$IMAGE=<filespec>
MENU$COMMAND=<filespec>
MENU$EXIT
```

where:

<form>	is a valid VAX FMS form name
<filespec>	is a valid VAX/VMS file specification

CHAPTER 4

RELEASE NOTES

This chapter contains information important to the installation and operation of BASEWAY.

4.1 FMS Startup Requirements

Due to the fact that several BASEWAY and PROGRAMMABLE DEVICE SUPPORT images are installed, the FMS shareable image (and error message file) must also be installed. This is not automatically done during installation, and many sites may not have made this change in the FMS startup file.

The following change must be made:

```
OLD:    $ !
        $ ! MCR INSTALL SYS$LIBRARY:FDVSHR/OPEN/SHARE
        $ ! MCR INSTALL SYS$MESSAGE:FDVSHR/OPEN/SHARE
        $ !

NEW:    $ !
        $ MCR INSTALL SYS$LIBRARY:FDVSHR/OPEN/SHARE
        $ MCR INSTALL SYS$MESSAGE:FDVMSG/OPEN/SHARE
        $ !
```

4.2 System Audit File

BASEWAY application events are logged in the System Audit file. This file is a circular file containing the last 1000 events that have been recorded. The advantage of a circular history file is that it needs no maintenance. Each application has its own System Audit file, normally created by DEFINEAPP.COM when the application is defined.

Events are recorded in the System Audit file by calling the BSL$LOG_EVENT service, as described in Section 4.21 of the BASEWAY System Programmer's Guide.

Some applications may log events so rapidly that 1000 events are only a small snapshot, and critical events are lost after only a few hours. Installations requiring a larger event file may create one with the following sequence of steps:

1. Run BSL$SYSTEM:UTLAPPSTP to shut down the application.

2. Create a File Description Language file for the history file:

```
$ ANALYZE/RMS/FDL/OUT=SYSAUD.FDL BSL$HISTORY_FILE
```

3. Edit the FDL file, and change the MAX_RECORD_NUMBER parameter to one more suited to your application:

```
$ EDIT/EDT SYSAUD.FDL
    1              IDENT  31-AUG-1984 17:06:28    VAX-11 FDL Editor
*F'MAX_RECORD_NUMBER'
*S/1000/5000/
  13                      MAX_RECORD_NUMBER       5000
*EXIT
DRA1:[KONKUS]SYSAUD.FDL;2 25 lines
```

4. Create a new System Audit file..

```
$ CREATE/FDL=SYSAUD.FDL
```

5. Restart your application.

Of course, since a new file has been created, all of the current audit trail has been lost.

On startup, the Event Logger process (EVENT_LOG) checks the first record of the System Audit file to see if it contains a valid header. If not, it preextends the event history file to its maximum length. If there are more than 10000 records, there will be a considerable delay before the "BASEWAY is running" message appears on the console terminal screen.

3 Supporting Multiple Applications

If more than one BASEWAY application is to communicate with a SHOP FLOOR GATEWAY system, then it is imperative that they use a common database. This entails changing the SYS$MANAGER:BSLSTART.COM command file to define the logical name BSL$SYSDATA so it points to a common directory for all applications. For example, if the database is to reside on node VAXA in the directory DRA0:[DATABASE], then all copies of BSLSTART.COM should define BSL$SYSDATA as follows:

```
BSL$SYSDATA = VAXA::DRA0:[DATABASE]
```

4.4 Using the Print Function

There are two possible print functions in BASEWAY and PROGRAMMABLE DEVICE SUPPORT: Print (Keypad 9) and Printall (PF1 (gold) Keypad 9).

The Print function routes an image of the current FMS screen to the session-default printer.

The Printall function lists all of the currently selected items to the default printer. For example, Printall used in conjunction with the LLT menu option (Display Ladder Listing) will cause the entire logic listing to be routed to the line printer.

The Print function is available for every screen in BASEWAY and PROGRAMMABLE DEVICE SUPPORT. Many of the screens also support Printall. The online help which is available for each menu option describes the capability of the various function keys.

APPENDIX A

SAMPLE INSTALLATION PROCEDURE

```
$ SET UIC [1,4]
$ SET DEFAULT SYS$UPDATE
$ @VMSINSTAL BSL010 MSA0:

        VAX/VMS Software Product Installation Procedure

It is 1-JUN-1984 at 12:05.
Enter a question mark (?) at any time for help.

* Are you satisfied with the backup of your system disk [YES]? YES

Please mount the first volume of the set on MSA0:.
* Are you ready? YES

%MOUNT-I-MOUNTED, BSL010 mounted on _MSA0:

The following products will be installed:

  BSL V1.0

        Beginning installation of BSL V1.0 at 12:07

%VMSINSTAL-I-RESTORE, Restoring product saveset A...

Previous logical name assignment replaced

BSLSTART.COM, the startup command procedure, is used to set up the
environment for the BASEWAY Application Bus.  During installation
it will be placed in the [SYSMGR] directory of the system root on
which this installation is being performed.  SYS$MANAGER:SYSTARTUP.COM,
your system startup procedure, should be modified to invoke this
procedure when the system boots.  However, it will not be necessary
to reboot the system after the installation, since this procedure
```

```
    invoked as part of the installation.

%VMSINSTAL-I-MOVEFILES, Files will now be moved to their target directories...

Installation Verification Procedure (IVP) starting

The installation verification of BASEWAY Application Bus v1.0 was successful.

        Successful installation of BSL V1.0 at 12:32

Enter the products to be installed from the next distribution volume set.
* Products [EXIT]: EXIT

        VMSINSTAL procedure done at 12:35
```

INDEX

November 1984

This manual describes BASEWAY concepts and features. Subroutine descriptions, syntax information, and other reference material are also included.

BASEWAY

System Programmer's Guide

SUPERSESSION/UPDATE INFORMATION:	This is a new document for this release.
OPERATING SYSTEM AND VERSION:	VAX/VMS V3.5
SOFTWARE VERSION:	BASEWAY Version 1.0
ORDER NUMBER:	

Digital Equipment Corporation
Manufacturing Field Application Center
24730 Crestview Court
Farmington Hills, Michigan 48018

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license. In addition, the following copyright notice must be included:

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10
VAX	VMS	SBI
DECnet	IAS	PDT
DATATRIEVE	TRAX	

CONTENTS

Contents

Contents

Contents

PREFACE

1.0 Manual Objectives

The BASEWAY System Programmer's Guide is intended to describe BASEWAY procedures and includes syntax information, error messages, and other reference information.

2.0 Audience

The intended audience for this manual is application programmers who have a basic knowledge of VAX/VMS and database concepts.

3.0 Prerequisites

The reader of this manual should have an understanding of the VAX/VMS operating system and an in-depth knowledge of at least one high-level programming language.

0 Structure of This Document

This manual is organized as follows:

Chapter 1: Describes the functions, applications, and organization of the BASEWAY system.

Chapter 2: Describes the purpose and format of interprocess messages.

Chapter 3: Introduces the system procedure descriptions detailed in Chapter 4.

Chapter 4: Describes selected BASEWAY subroutines.

Appendix A: Summarizes procedure parameter notation for BASEWAY calls.

Appendix B: Explains the purpose, capabilities, and structure of the SHOP FLOOR GATEWAY.

Appendix C: Provides a detailed description of the use of Device Interface Modules (DIMs) including an example DIM.

Appendix D: Gives information relevant to adding new device support.

Appendix E: Glossary.

5.0 Associated Documents

Further information on various topics covered in this manual may be found in the following manuals:

- RSX-11M/M-Plus Executive Reference Manual
  (order number AA-L675A-TC)
- RSX-11M/M-Plus Guide to Writing an I/O Driver
- RSX-11M/M-Plus System Management Guide
  (order number AA-L679B-TC)
- RSX-11M/M-Plus Task Builder Manual
  (order number AA-L680A-TC)
- VAX ALL-IN-1 Application Programmer's Reference Guide
  (order number AA-N324A-TE)
- BASEWAY Installation Guide/Release Notes
  (order number XX-12345-01)
- BASEWAY User's Manual and Utilities Guide
  (order number XX-12346-01)
- SHOP FLOOR GATEWAY Installation Guide/Release Notes
  (order number XX-12355-01)
- PROGRAMMABLE DEVICE SUPPORT Installation Guide/Release Notes
  (order number XX-12365-01)
- PROGRAMMABLE DEVICE SUPPORT User's Manual and Utilities Guide
  (order number XX-12367-01)

- VAX DATATRIEVE User's Guide
  (order number AA-K079A-TE)
- VAX FMS Form Driver Reference Manual
  (order number AA-L319A-TE)
- VAX FMS Language Interface Manual
  (order number AA-N209A-TE)
- VAX FMS Utilities Reference Manual
  (order number AA-L320A-TE)
- VAX LINKER Reference Manual
  (order number AA-D019C-TE)
- VAX PL/I Encyclopedic Reference
  (order number AA-H952A-TE)
- VAX PL/I User's Guide
  (order number AA-H951A-TE)
- VAX Run-Time Library User's Guide
  (order number AA-L824A-TE)
- VAX Utilities Reference Manual
- VAX/VMS Command Language User's Guide
  (order number AA-D023B-TE)
- VAX/VMS System Services Reference Manual
  (order number AA-D018C-TE)

Preface

CHAPTER 1

INTRODUCTION TO BASEWAY

1.1 Overview

The BASEWAY system provides tools for the development and control of complex manufacturing applications where accurate and timely communication with shop floor devices is vital. These tools can reduce application development and maintenance time by replacing significant amounts of application control and programmable device-specific communications code.

The BASEWAY product works together with DIGITAL's SHOP FLOOR GATEWAY product. While BASEWAY provides application program communications and control functions, the SHOP FLOOR GATEWAY provides the actual communications interface to shop floor devices.

Programmable devices are discrete processors that are used in shop floor control and data acquisition applications. These devices include programmable controllers, numerical controllers, robots, bar code scanners, and many others. Each of the devices involved with an application is defined to BASEWAY and identified by a unique device name.

Because programmable device access subroutines allow the user to write an application program in a device-independent fashion, the program is not tied to a programmable device or vendor. It can therefore be upgraded, usually by a simple redefinition of the programmable device to BASEWAY. See Appendix B for a discussion of the GATEWAY and Appendix D for information about adding new device support.

A wide range of manufacturing applications can be developed and controlled with BASEWAY, including inventory control, part tracking, and quality control.

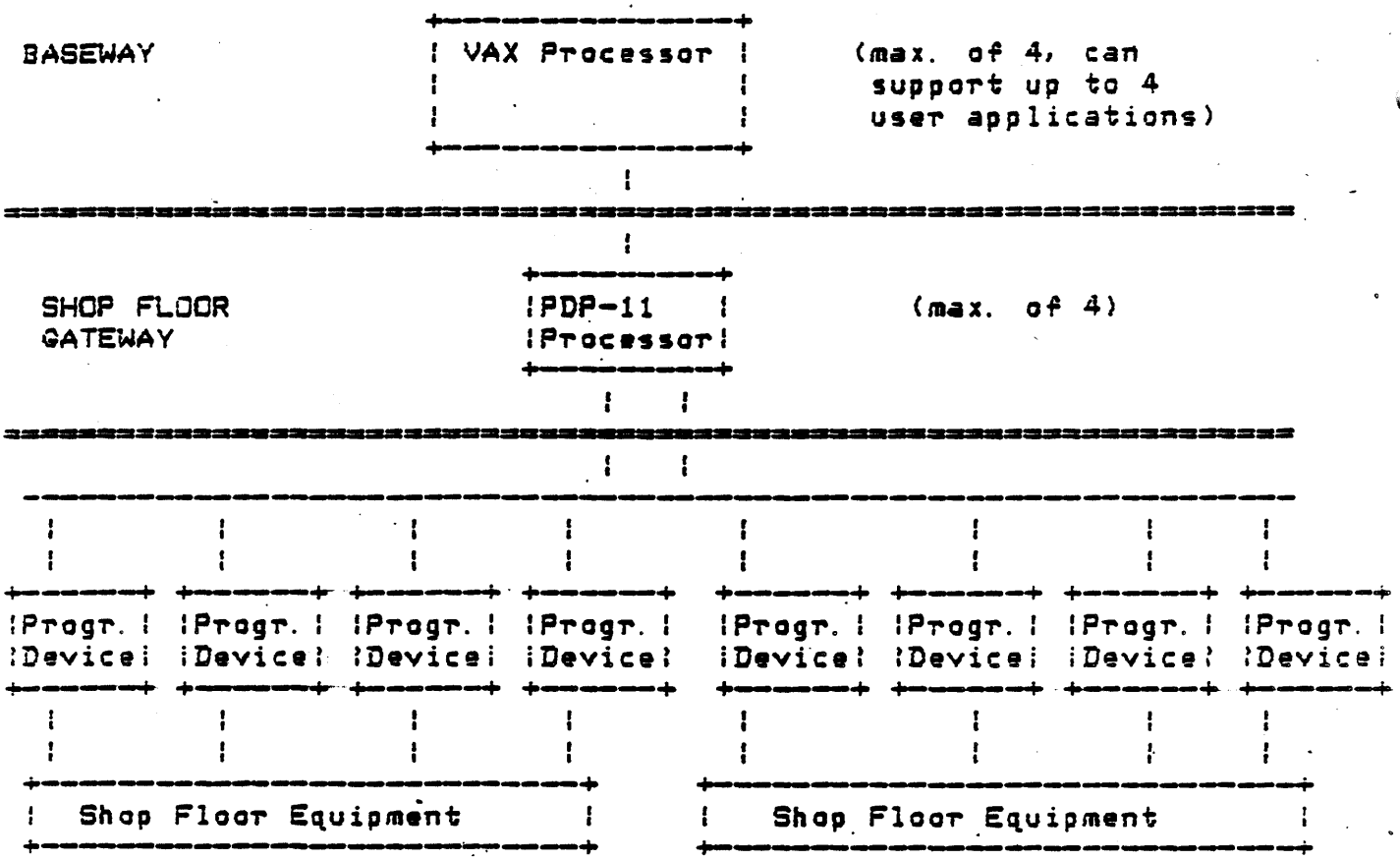


Figure 1. System Overview

1.2 Facilities

1.2.1 Programmable Device Access

BASEWAY data structures, when used by an application program to access devices, can determine device status and read and write data to devices.

1.2.1.1 Programmable Device Definition -

Programmable devices are defined interactively using the BASEWAY configuration editor. Attributes assigned to each programmable device include:

- name

- o description or comments
- o manufacturer name and model number
- o communications network address
- o memory size
- o physical location
- o device type and status
- o date of installation

These assigned attributes are checked for validity when the device is defined. Up to 2000 programmable devices may be defined.

1.2.1.2 Shop Floor Data Definition -

Individual pieces of data in a programmable device are defined interactively with the BASEWAY configuration editor. Such data items are termed "known points". Any address in a programmable device may be read or written when referred to by address, but known points may be referred to by point name.

When known points are defined to the BASEWAY system, various attributes are assigned to that data. Attributes assigned to each known programmable device address include:

- o address of data
- o format of data
- o associated equipment name
- o name of each data point
- o minimum sample time
- o unit descriptions
- o destination application and process name

Several data formats are supported. These include:

- o status conditions (single bit values)

- 12 bit BCD values
- 16-bit BCD values
- 16-bit signed binary values

Known points may be defined as monitored, controlled, or both. Monitored points are sampled at user-specified intervals by the SHOP FLOOR GATEWAY. Controlled points are known points that are written to by application programs.

1.2.1.3 Programming Interface -

The BASEWAY programming interface is the means by which applications programs communicate with each other and with shop floor devices. The programming interface permits application programs to interact with shop floor devices in a variety of ways:

- Generic Access allows an application program to perform primitive functions through device-independent routines. These functions include reading and writing into device addresses, starting and stopping devices, and getting device attributes.
- Polled Access permits the SHOP FLOOR GATEWAY to periodically sample device data and send a message to an application program when data changes.

1.2.2 Messaging/Networking

BASEWAY software designers may divide complex systems into functionally separate subsystems called "applications". Up to four of these applications may be defined to BASEWAY. These may reside on a single VAX CPU or on separate CPUs configured in a DECnet network. A global database contains all programmable device definitions, and programs in each application can access a device simultaneously. The product can concurrently support up to four gateways.

Programs communicate with each other through a messaging facility. Through "named message ports", a program can communicate with another program, even if they are not on the same CPU.

1 2.3 Application Control

The Application Control facility provides a controlled environment for application startup and shutdown. During application startup, a script file is executed. This file may create logical names and mailboxes, start application programs, or perform DCL commands. An application program may be monitored or unmonitored. If a monitored application program fails, the BASEWAY Application Control facility logs the event and begins an orderly shutdown of all other application processes.

1.2.4 Session Control

The BASEWAY Session Control facility allows a user at a VT100 or VT200 series terminal to select from a menu of options and move from one application program to another. Each menu item may invoke another menu screen, an application program, or a command file. A sample menu structure is provided with the product. This structure may be customized on a site-specific basis by using the VAX FMS Forms Editor. User-written application programs may be added to the menu structure.

1.2.4.1 User Definition -

Each user is defined to BASEWAY. A user definition contains demographic data, user-customization data, privileges, and application-specific data. The specific attributes that may be defined for each user include:

- user name
- user's VMS user name
- nickname
- address
- telephone number
- title or position
- department
- default line printer device
- password for user verification

- o initial menu form and form library
- o privilege masks

1.2.4.2 Terminal Definition -

Each terminal device is defined to BASEWAY. A terminal definition contains privileges and application-specific data. If both a terminal definition and a user definition exist for the current session, the terminal definition overrides the user definition. The attributes that may be defined for each terminal include:

- o terminal device name
- o physical location
- o default line printer device
- o initial menu form and form library
- o privilege masks

1.2.5 Audit Trail

The BASEWAY Audit Trail records system events about user logins, task selection, programmable device events, and other events. An associated report facility can be used to view the information, providing specific information about users and devices.

1.3 Functions

The BASEWAY system consists of a group of detached processes that are initially created when an application system is "started".

The processes that are normally active and running on the system are:

Event Processor (EVENT_PROC)
Event Logger (EVENT_LOG)
Network Interface (NET_INTER)
Gateway Initializer (GATE_INIT)
User Application Processes
 (sample Data Processor (DATA_PROC)
  included in basic system)

A set of subroutines in the system library performs all of the necessary overhead of sending messages. These routines assign mailbox channels, format interprocess messages, and send appropriate data.

1.4 Applications

Up to four (4) applications, running on from one to four VAX processors, are supported. Each application is assigned a unique VAX/VMS group number, and all user programs running in the same application are assigned to the same group. Applications are defined by unique names.

1.5 Access to BASEWAY Processes

1.5.1 Interprocess Messages

Generally, the programs running under the BASEWAY communicate among each other via mailboxes. Data is transmitted in data packets, also referred to as "interprocess messages." A more complete explanation of interprocess messages, including their formats, is given in Chapter 2.

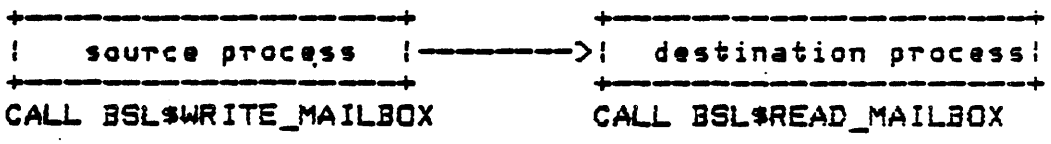


Figure 2. Interprocess Messages Between Programs

If the programs are not running under the same application, they n still use the same mechanism to communicate with each other. The .SEWAY network interface programs handle all routing of the messages.

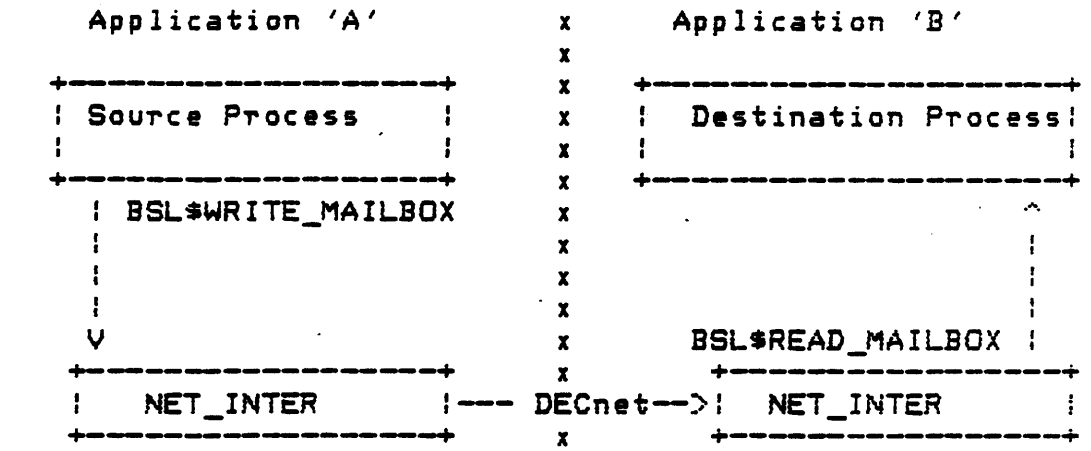


Figure 3. Interprocess Message Communication Between Applications

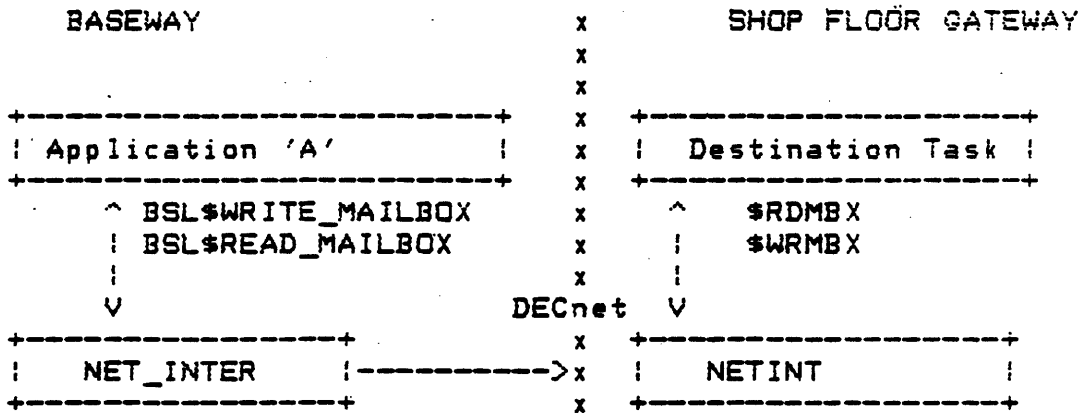


Figure 4. Interprocess Message Communication Between Application and Gateway

1.6 Data Files

VAX-RMS data files are used to maintain configuration and history data for the BASEWAY. These files must be present at all times. They are:

BSL$HISTORY_FILE	Circular file containing a chronological history of BASEWAY events.
BSL$SYSTEM_FILE	Indexed file containing definitions of all applications, device sets, and gateways.
BSL$DEVICE_FILE	Indexed file containing all programmable device definitions.
BSL$POLLING_FILE	Indexed file describing sets of registers on a programmable device that are to be collected together.
BSL$REGISTER_FILE	Indexed file describing all polled registers for each of the programmable devices.
BSL$ENTITY_FILE	Indexed file containing definitions of all of the types of data that are collected from programmable devices.
BSL$TERMINAL_FILE	Indexed file containing menu information for a particular terminal.
BSL$USER_FILE	Indexed file containing session control information for a particular user.

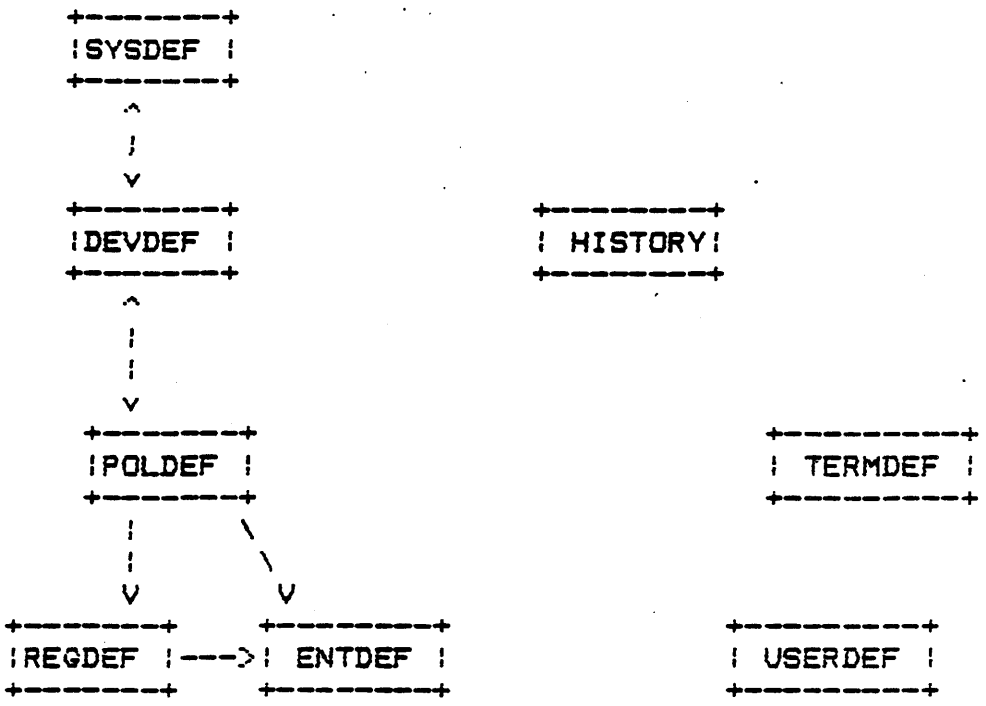


Figure 5. BASEWAY Data Files

1.7 BASEWAY Processes

Processes on BASEWAY may be pictured as shown below:

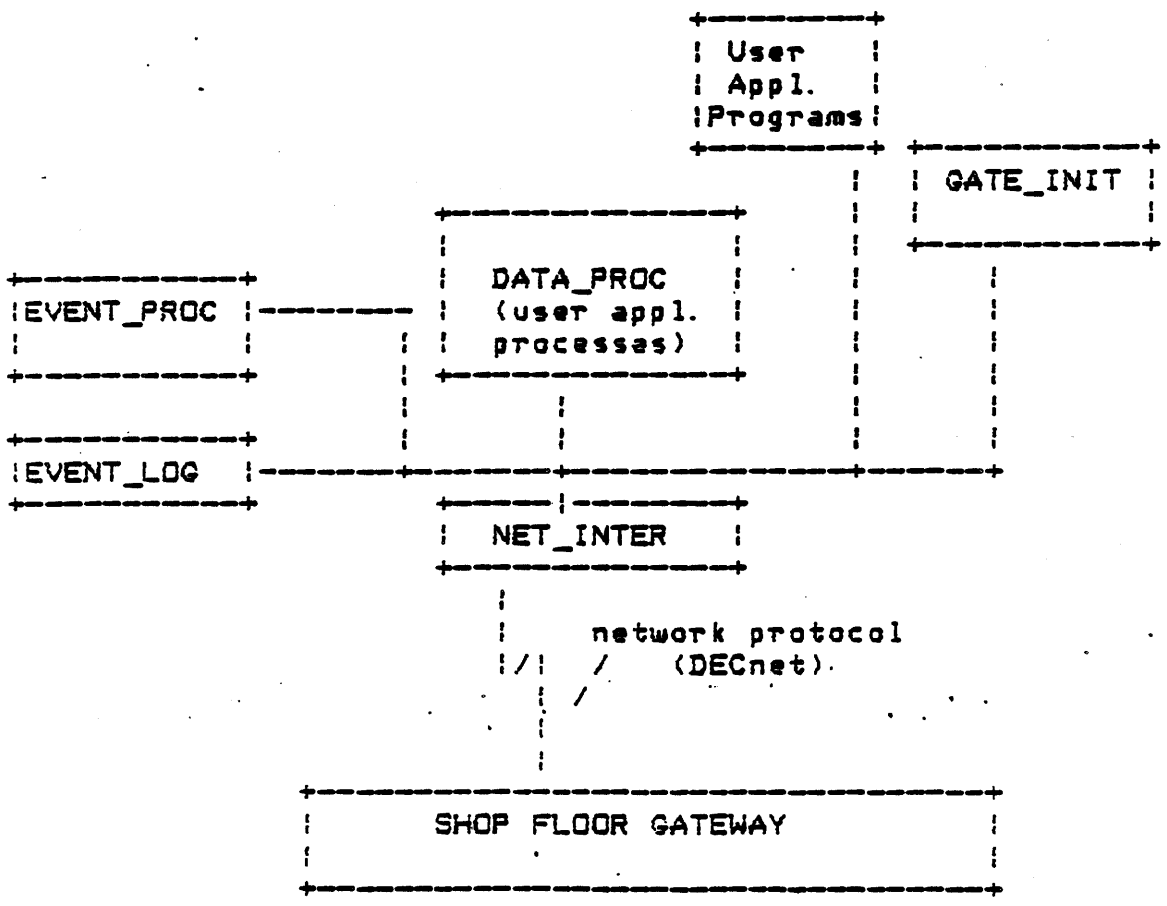


Figure 6. Diagram of BASEWAY Processes

All application programs communicate with the SHOP FLOOR GATEWAY via the NET_INTER program. Any program may communicate directly with any other program.

1.8 Event Processor (EVENT_PROC)

All other BASEWAY processes run as subprocesses to the Event Processor. EVENT_PROC is driven at system startup time by a command file which tells it how to start up the other processes, as well as creating various group logical names, global sections, etc. It is also responsible for reporting and handling significant external events, such as a link failure with a GATEWAY system and PD timeout.

The Event Processor is responsible for the following functions:

- o subprocess termination notice
- o gateway failure reporting
- o gateway event reporting
- o establishing group logical names for application-specific databases
- o establishing group-accessible global sections for application-specific databases.

1.9 Event Logger (EVENT_LOG)

The EVENT_LOG process receives system events messages from other processes through its process mailbox and logs them to various operator terminals, the system event log file, the VAX operator's console, or the console of the SHOP FLOOR GATEWAY.

This process is always running on BASEWAY. Logs to 1K record circular file; can change this size

1.10 Network Interface (NET_INTER)

The primary function of this process is the routing and delivery of interprocess mail messages. The process receives messages from its mailbox or over a logical link from a gateway. It then sends the messages to another process's mailbox or to the Network Interface via a gateway.

This is part that knows node names.
Other parts DO NOT CARE — they use appl. bus. names
This lends itself to failover (not yet automatic)

1.11 Gateway Initialization (GATE_INIT)

This process continuously monitors the activity of the SHOP FLOOR GATEWAY systems and responds to significant gateway events.

This process controls the sequence of events which occur when a gateway is booted (downline system loaded). It exchanges various messages with the TSKWCH and GATEVP tasks in the SFQ to start various gateway tasks running and load the polling database.

1.12 User Application Processes and Data Processor (DATA_PROC)

These processes receive most of the data that is sent from the gateway as a result of device polling. They are responsible for formatting the data and notifying an application data processor when the data arrives. An example DATA_PROC is included with the basic system. Need one per application; Needed only for appl. using automatic PC polling.

1.12.1 Disposition of Polled Data

DATA_PROC acts rather like a "sink" for polled data. In fact, many different polled data receivers may be run as part of an application. For example, one process could receive quality data, another equipment fault data, etc.

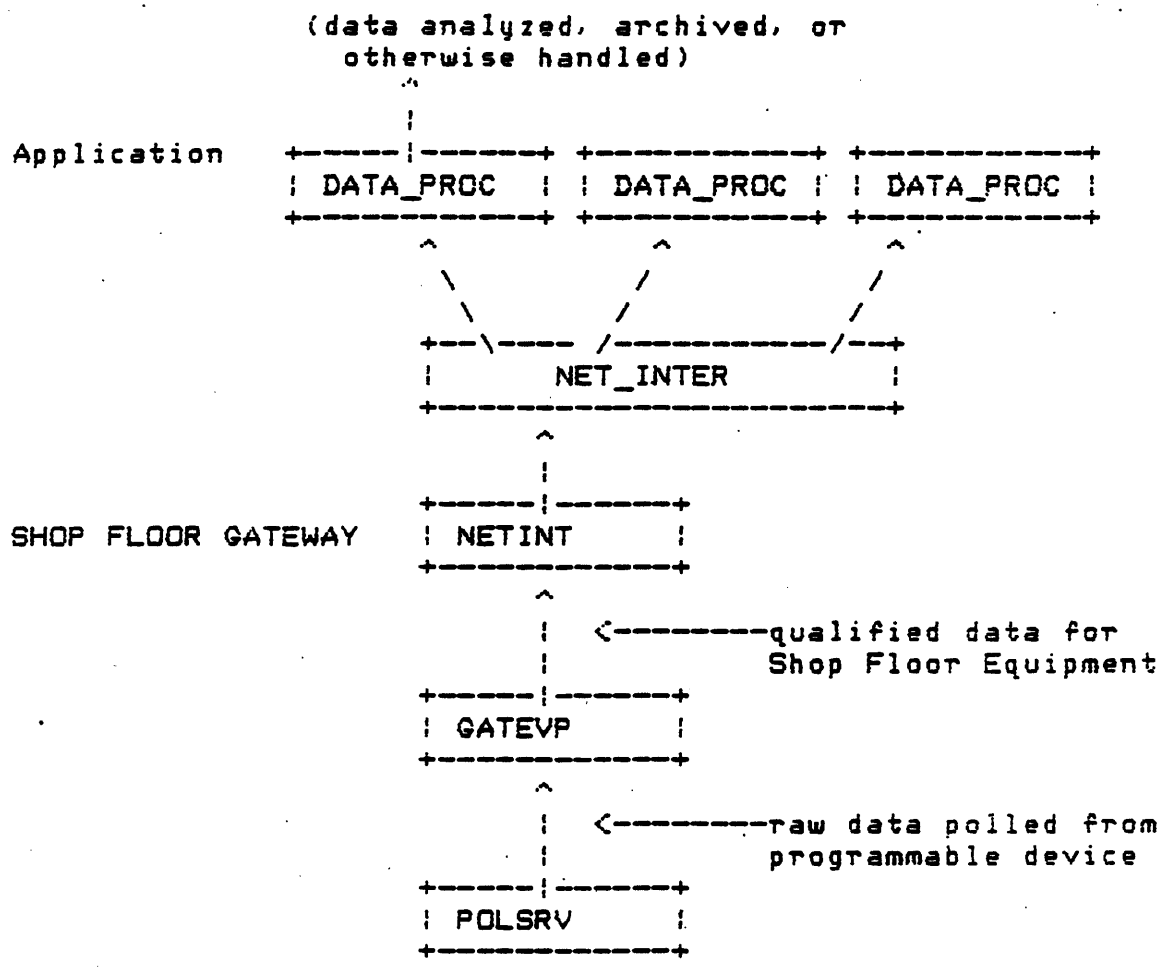


Figure 7. Polled Data Receivers for One Application

1.12.2 Polled Data

Application programs running on BASEWAY normally send interprocess messages to the SHOP FLOOR GATEWAY, although these messages may originate from a task in the gateway itself. (See Chapter 2 for additional information on interprocess messages.)

1.13 Message Data Types

Data types (dtype) fall into three categories: atomic, string, and miscellaneous.

1.13.1 Atomic

Atomic data types used in BASEWAY messages are defined and encoded as follows:

DSC_K_DTYPE_Z	0	Unspecified. The calling program has specified no dat type. The receiving procedure should assume that the data is of the corre type.
DSC_K_DTYPE_B	6	Byte Integer. 8-bit signed 2s complement integer.
DSC_K_DTYPE_W	7	Word Integer. 16-bit signed 2s complement integer.
DSC_K_DTYPE_L	8	Longword Integer. 32-bit signed 2s complement integer.
DSC_K-DTYPE_F	10	F_Floating. 32-bit F_floating quantity representing a single-precision number.
DSC_K_DTYPE_D	11	D_Floating. 64-bit D_floating quantity representing a double-precision number.

13.2 String

String data types used in BASEWAY messages are defined and encoded as follows:

DSC_K_DTYPE_T	14	Character-Coded Text. A single 8-bit character (atomic data type) or a sequence of 0 to 65536 8-bit characters (string data type).
DSC_K_DTYPE_V	1	Bit. An aligned bit string. A string of 0 to 65536 contiguous bits. The first bit is bit 0 of the first byte and the last bit is any bit in the last byte. Remaining bits in the last byte must be zero on read and are cleared on write.

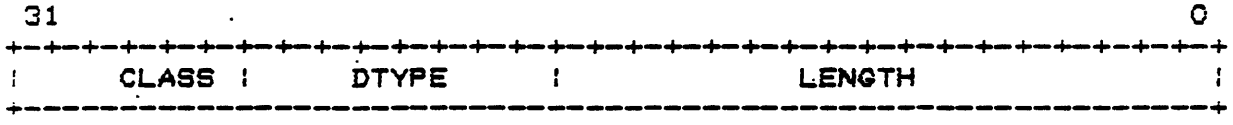
1.13.3 Miscellaneous

Miscellaneous data types used in BASEWAY messages are defined and encoded as follows:

DSC_K_DTYPE_DSC	24	Descriptor. This data type allows a descriptor to be a data type; thus, levels of descriptors are allowed.

1.14 Message Descriptor Prototype

Each class of descriptors consists of at least one longword in the following format:

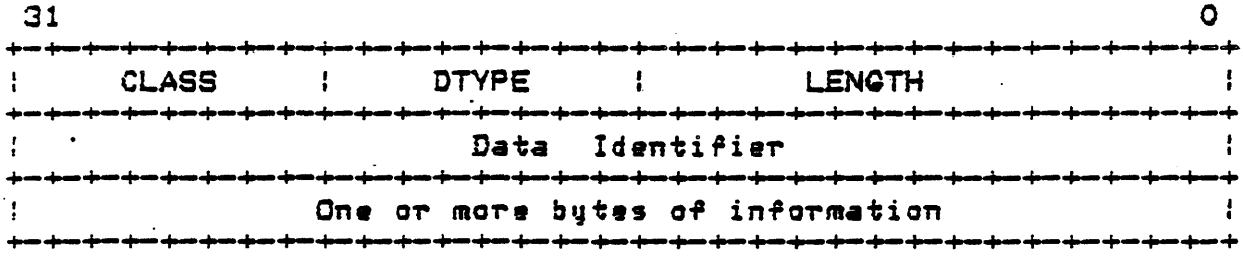


Symbol	Description

DSC_W_LENGTH <0,15:0>	A one-word field specific to the descriptor class, typically a 16-bit (unsigned) length.
DSC_B_DTYPE <0,23:16>	A one-byte data type code.
DSC_B_CLASS <0,31:24>	A one-byte descriptor class code.

1.14.1 Scalar, String Descriptor (DSC_K_CLASS_MSGS)

A single descriptor form is used for scalar data and fixed-length strings.



Symbol	Description
DSC_W_LENGTH	Length of data in bytes, unless the DSC_B_DTYPE field contains the value 1 (bit). Length of data item is in bits for bit.
DSC_B_DTYPE	A one-byte data type code.
DSC_B_CLASS	192 = DSC_K_CLASS_MSGS
DSC_L_DATAID	A longword containing the data identifier value. This value uniquely names the described data field.
DSC_R_DATA	Start of the data field.

14.2 Example DATA_PROC

An example DATA_PROC is included in Version 1.0 of BASEWAY and is started by a default startup file. It logs an event through the Event Logger each time a polled data message is received. If polled data is to be used as part of the application, this example may be useful as a starting point for creating your own polled data receiver process. The source for this example is in BSL$ROOT:[SOURCE.DATAPROC].

CHAPTER 2

INTERPROCESS MESSAGES

2.1 Overview

All BASEWAY programs communicate via Interprocess Messages. These messages are used to exchange data and request command functions.

Interprocess messages consist of two distinct areas: a fixed message header and an optional, variable-length data area. The header contains routing information for the message and a message identification code. The size and format of the optional data area is dependent on the message identification code.

The generic format of an interprocess message is diagrammed below:

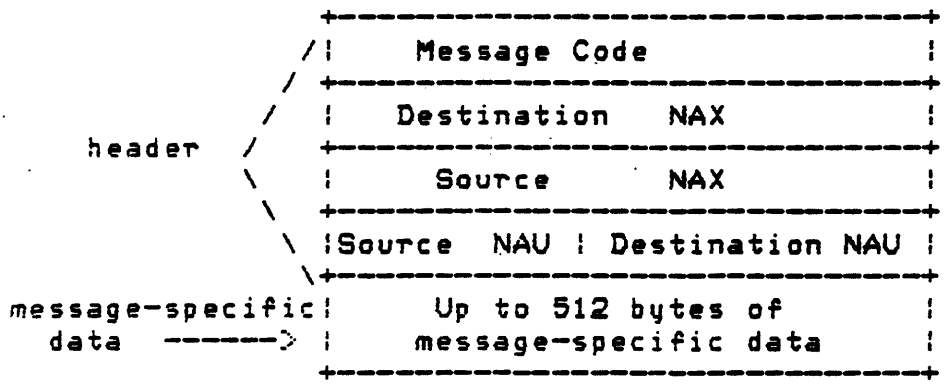


Figure 8. Generic Interprocess Message Format

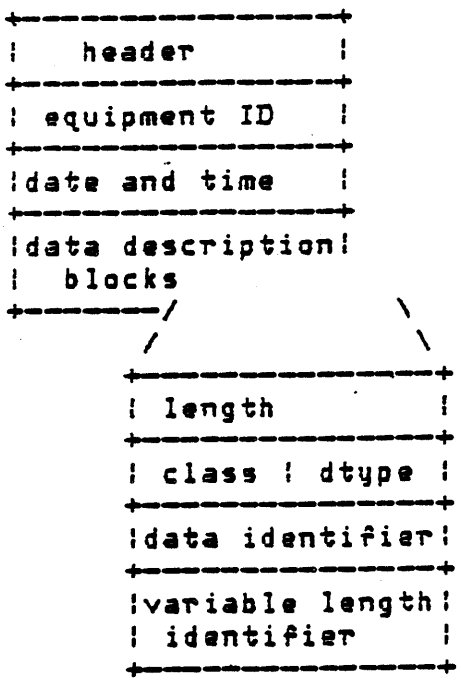
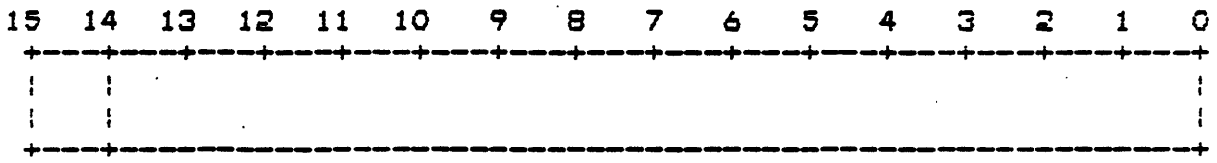


Figure 9. Encoding of an Interprocess Message

2.1.1 Message Code

A message code is a word which indicates consent or action to be taken as a result of the message. Message codes 1--32767 are reserved for Digital Equipment Corporation.

2.1.1.1 Format of Message Code -



Bits 0--14 identify message code
Bit 15 0, if Digital Equipment message
1, if user message

2.1.2 Source and Destination NAX

Each application, device set, and gateway is assigned a unique 16-bit integer number at definition time. This number is used to route messages and transactions between systems.

2.1.3 Source and Destination NAU

Each program in an application can have a Network Addressable Unit (NAU) associated with it. An NAU, along with a NAX, indicates a unique process in a network.

There are two types of NAUs: temporary and permanent. Temporary NAUs are in the range -1 to -127, and are assigned by calling the BSL$CREATE_PORT routine. Permanent NAUs are created by assigning logical names and creating mailboxes in the EVENT_PROC startup file. Permanent NAUs are in the range 1 to 127:

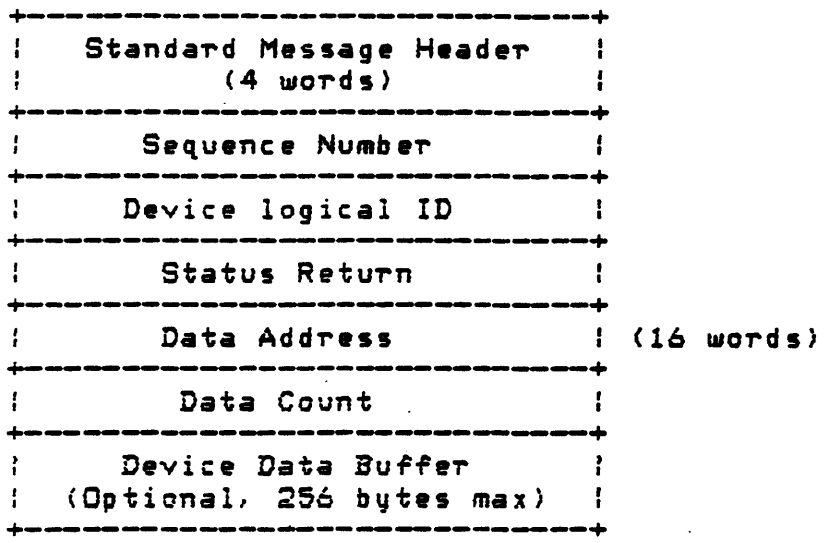
1--63 Reserved for Digital Equipment
64--127 Available to user

2.1.4 Message-Specific Data

The message-specific data contains information related to the individual message. The following sections will define the message-specific data formats for each message code.

2.2 Do Generic I/O Request

Message format:



Message Codes:
MSG_GENERIC_READ
MSG_GENERIC_WRITE
MSG_GENERIC_WRITEVFY
MSG_GENERIC_STARTDEV
MSG_GENERIC_STOPDEV
MSG_GENERIC_READDEVSTAT
MSG_GENERIC_LOGONDEV
MSG_GENERIC_LOGOFFDEV
MSG_GENERIC_STARTUPLOAD
MSG_GENERIC_ENDUPLOAD
MSG_GENERIC_STARTDOWNLOAD
MSG_GENERIC_ENDDOWNLOAD

Sent to: This kind of message is routed to GENSRV. It causes GENSRV to format and execute the request.

Sequence Number - may be set by the requesting program. The GENSRV device request always returns a response message. The response will contain the sequence number of its command, making this field usable as a synchronization aid or "sanity check."

Device Logical ID - an unsigned word value that is uniquely associated with each programmable device defined.

Status Return - not used.

Data Address - the starting memory address in the programmable device.

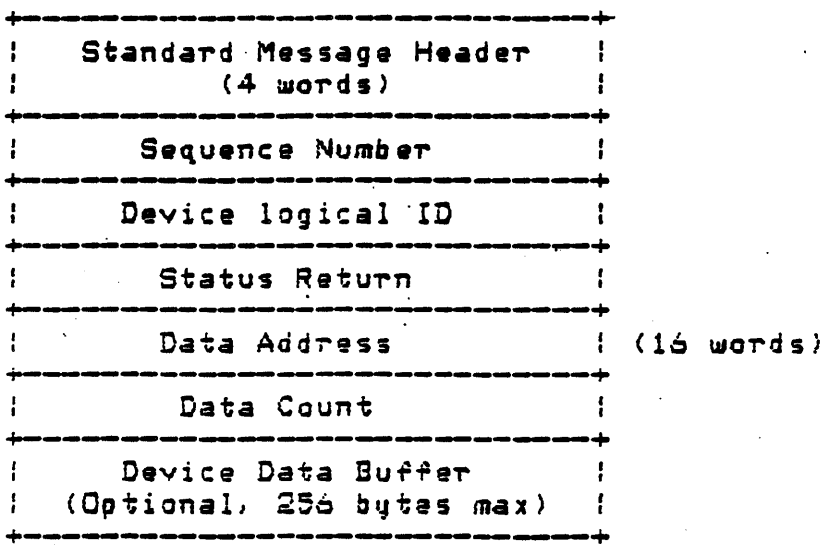
Data Count - the number of memory locations that are to be transferred.

Device Data Buffer (optional) - variable length.

Device Data Buffer (optional) - variable length.

2.3 Do Generic I/O Response

Message format:



Sent to: This is sent to GENSRV and contains a status code and any data that was requested.

Sequence Number - the sequence number of its command message.

Device Logical ID - unchanged.

Status Return - the status of the transaction. A value of 1 indicates success.

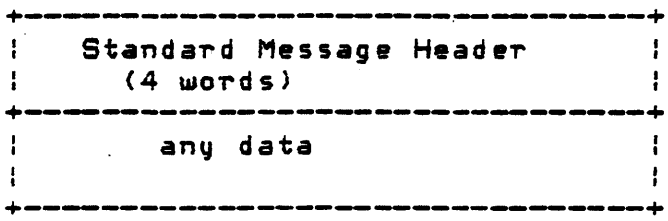
Data Address - unchanged.

Data Count - unchanged.

Device Data Buffer (optional) - variable length.

2.4 Gateway Loopback

Message format:

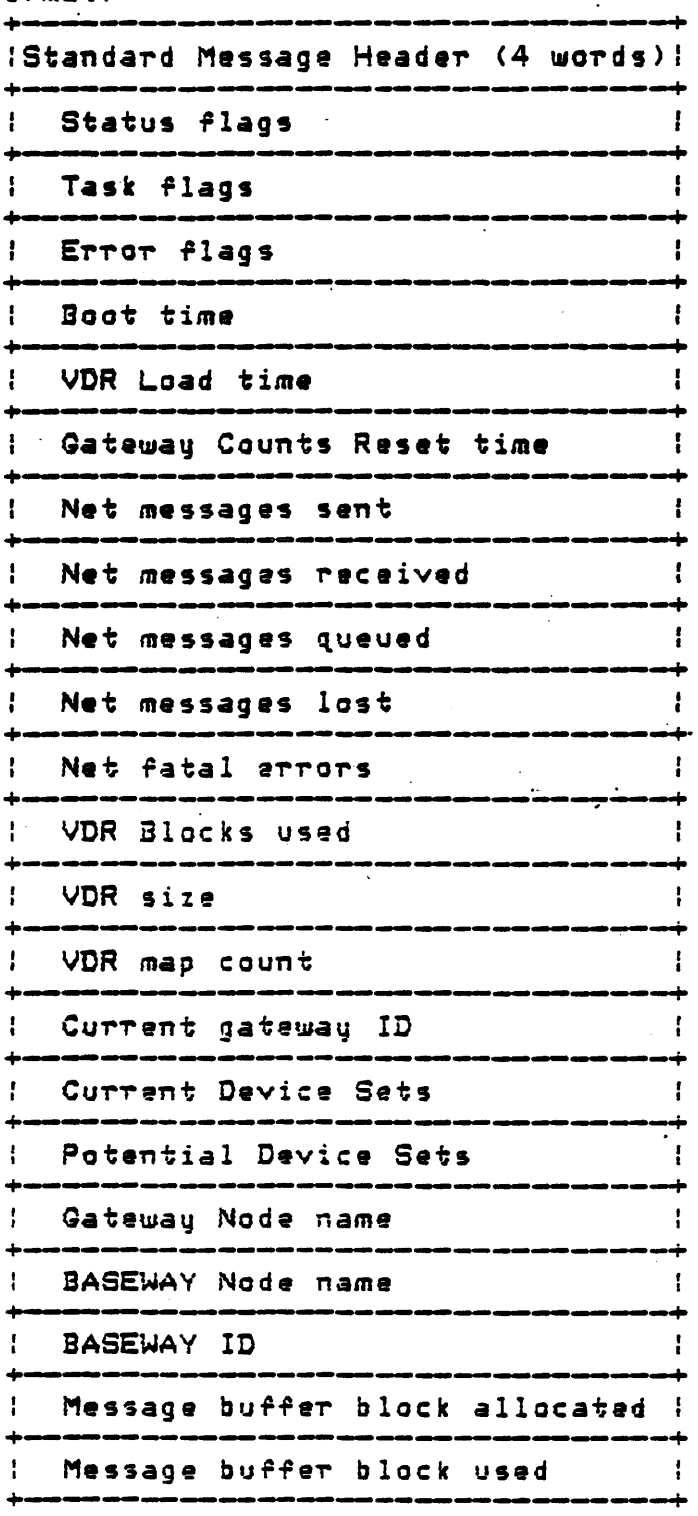


Sent to:

This message is sent to NETINT on the gateway.

2.5 Get Gateway Status

Message format:

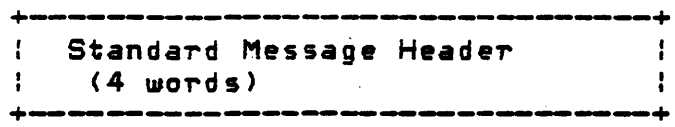


Sent to:

This is sent to GATEVP.

2.6 Get Network Status for Gateway

Message format:

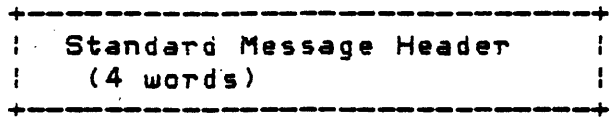


Sent to:

This message is sent to NETINT on the SHOP FLOOR GATEWAY.

2.7 Get Polled Device Statistics

Message format:

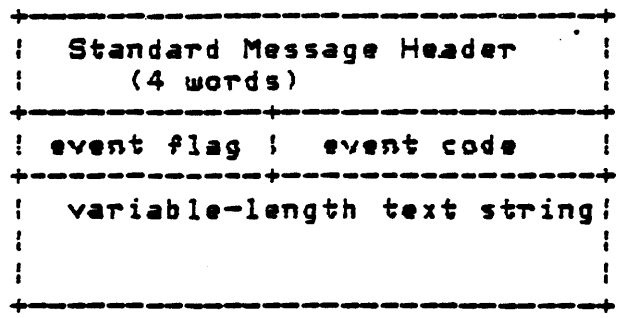


Sent to:

This message is routed to POLSRV.

2.8 Log Event

Message format:



Sent to:

This is sent to EVENT_LOG to initiate event logging.

2.9 Reload VDR

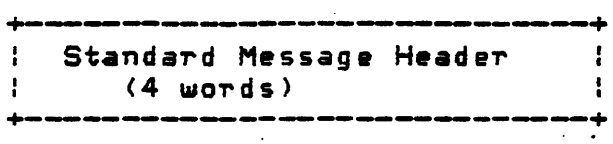
Message format:

Sent to:

This is sent to GATEVP.

2.10 Reset Network Counts

Message format:

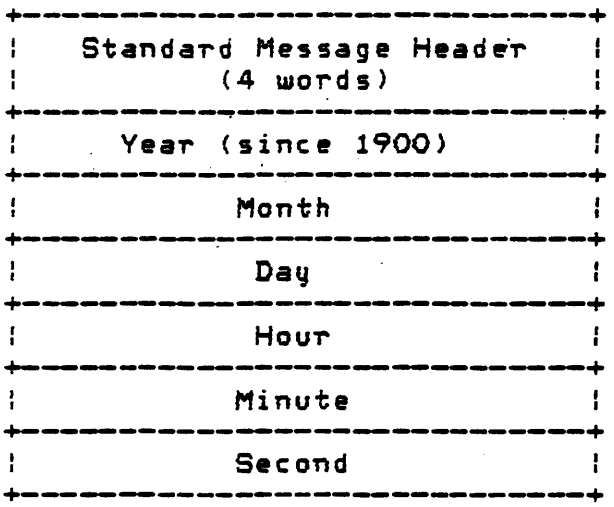


Sent to:

This message is sent to NETINT.

2.11 Set Gateway Time

Message format:

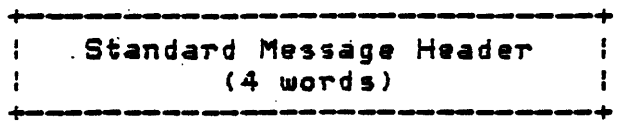


Sent to:

This message is sent to TSKWCH and causes it to set the RSX-11S system time to the time specified by the message.

2.12 Shutdown Application

Message format:

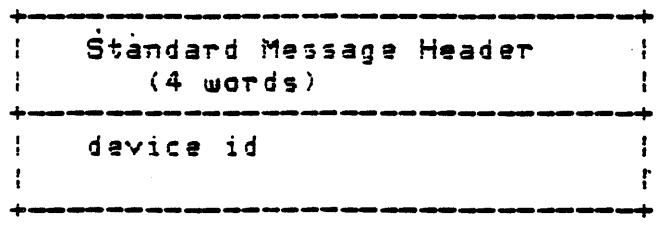


Sent to:

This is normally sent to EVENT_PROC to effect system shutdown.

2.13 Start Polling on a Device

Message format:

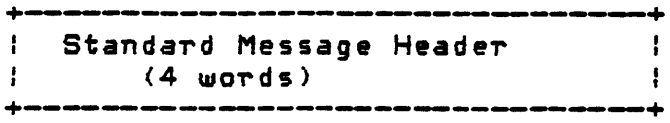


Sent to:

This message goes to POLSRV.

2.14 Stop Gateway

Message format:

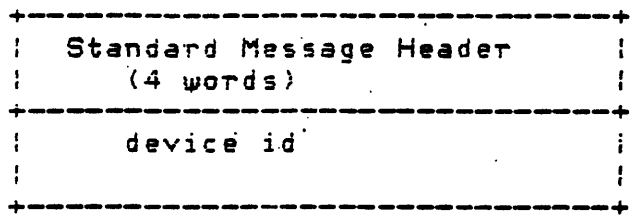


Sent to:

This is sent to TSKWCH.

2.15 Stop Polling on a Device

Message format:



Sent to:

This message is routed to POLSRV.

CHAPTER 3

INTRODUCTION TO SUBROUTINE DESCRIPTIONS

Each high-level language supported by VAX/VMS provides some mechanism for calling an external procedure and passing arguments to that procedure. The actual mechanism and terminology used, however, vary from one language to another. Since it is not possible to describe the ways in which each high-level language calls system services, for specific information, it is recommended that you refer to the appropriate high-level language user's guide.

VAX/VMS system services are external procedures that accept arguments. There are three ways to pass arguments to system services:

- by immediate value. The argument is the actual value to be passed (a number or a symbolic representation of a numeric value).
- by address (also called "by reference"). The argument is the address of an area or field that contains the value. An argument passed by address is usually expressed as a reference name or label associated with an area or field. (In fact, one common error is to pass a numeric value without indicating that it is passed by value; if the compiler assumes the numeric value is an address, a run-time "access violation" error occurs when, for example, the image tries to access virtual address 0 or 1.)
- by descriptor. This argument is also an address, but of a special data structure called a character string descriptor.

A description of each service is given in Part II of the VAX/VMS Services Reference Manual and indicates how each argument is to be passed. Phrases such as "an address" and "address of a character string descriptor" identify address and descriptor arguments, respectively. Words like "indicator," "number," "value," or "mask" indicate an argument passed by immediate value.

Some services also require service-specific data structures that indicate functions to be performed or hold information to be returned. You can use this information and information from your programming language manuals to define such an item list.

3.0.1 Testing Return Status Codes in High-Level Languages

When a service returns control to your program, it places a return status value in the general register R0. The value in the low-order word indicates either that the service completed successfully or that some specific error prevented the service from performing some or all of its functions. After each call to a system service, you must check to see whether it completed successfully. You can also test for specific error conditions.

Each language provides some mechanism for testing the return status. Often you need only check the low-order bit, for example, with a test for TRUE (success or informational return) or FALSE (error or warning return).

To check the entire value for a specific return condition, each language provides a way for your program to determine the values associated with specific symbolically defined codes. You should always use these symbolic names when you write tests for specific conditions.

The following chapter describes selected BASEWAY procedures. These procedures are presented by category and in alphabetical order by entry name.

BASEWAY subroutines may be included in a user program by LINKing with the library, BSL$LIBRARY.

3.0.2 Compiling and Linking a VAX PL/I Program

The text library BSLDEF.PLI contains a variety of codes and routine definitions for BASEWAY. It includes most of the definitions that you will need for writing your own PL/I application programs. BSLDEF.PLI can be found in the directory BSL$LIBRARY.

The following commands will compile and link a VAX PL/I program:

```
$ PLI MYTEST
$ LINK MYTEST, SYS$LIBRARY:BSLLIB/LIB
```

3.0.3 Compiling and Linking a VAX BLISS-32 Program

The include file BSLDEF.REQ contains a variety of codes and routine definitions for BASEWAY. It includes most of the definitions that you will need for writing your own application programs. BSLDEF.REQ can be found in the directory BSL$LIBRARY.

The following commands will compile and link a VAX BLISS-32 program:

```
$ BLISS MYTEST
$ LINK MYTEST, SYS$LIBRARY:BSLLIB/LIB
```

3.0.4 Compiling and Linking a VAX FORTRAN Program

The text library BSLDEF.FOR contains a variety of codes and routine definitions for BASEWAY. It includes most of the definitions that you will need for writing your own FORTRAN application programs. BSLDEF.FOR can be found in the directory BSL$LIBRARY.

The following commands will compile and link a VAX FORTRAN program:

```
$ FORTRAN MYTEST
  LINK MYTEST, SYS$LIBRARY:BSLLIB/LIB
```

3.0.5 Compiling and Linking a VAX BASIC Program

The include file BSLDEF.BAS contains a variety of codes and routine definitions for BASEWAY. It includes most of the definitions that you will need for writing your own BASIC application programs. BSLDEF.BAS can be found in the directory BSL$LIBRARY.

The following commands will compile and link a VAX BASIC program:

```
$ BASIC MYTEST
$ LINK MYTEST, SYS$LIBRARY:BSLLIB/LIB
```

3.0.6 Compiling and Linking a VAX C Program

The include file BSLDEF.H contains a variety of codes and routine definitions for BASEWAY. It includes most of the definitions that you will need for writing your own C application programs. BSLDEF.H can be found in the directory BSL$LIBRARY.

The following commands will compile and link a VAX C program:

```
$ CC MYTEST
$ LINK MYTEST, SYS$LIBRARY:BSLLIB/LIB
```

3.0.7 Compiling and Linking a VAX COBOL Program

The copy file BSLDEF.LIB contains a variety of codes and routine definitions for BASEWAY. It includes most of the definitions that you will need for writing your own COBOL application programs. BSLDEF.LIB can be found in the directory BSL$LIBRARY.

The following commands will compile and link a VAX COBOL program:

```
$ COBOL MYTEST
$ LINK MYTEST, SYS$LIBRARY:BSLLIB/LIB
```

3.0.8 Compiling and Linking a VAX PASCAL Program

The definition file BSLDEF.PAS contains a variety of codes and routine definitions for BASEWAY. It includes most of the definitions that you will need for writing your own Pascal application programs. BSLDEF.PAS can be found in the directory BSL$LIBRARY.

The following commands will compile and link a VAX PASCAL program:

```
$ PASCAL BSLDEF/ENVIRONMENT
$ PASCAL MYTEST
$ LINK MYTEST, BSLDEF, SYS$LIBRARY:BSLLIB/LIB
```

3.1 How To Use Procedure Descriptions in This Manual

Each procedure description consists of the following categories. Certain procedures have an additional "Notes" section.

NOTE: All of the procedures described in this document return a completion status condition code as a function return value.

Calling Format:

This section shows the entry name and a generalized format for calling the procedure from a high-level language, with all arguments listed in positional order. Spaces between arguments are present for readability and are not part of the statement syntax.

Arguments:

This section describes each of the arguments in detail, along with any special argument-passing requirements.

Return Status:

This section lists the possible error return status codes from the procedures, with an explanation of the return condition. The returns are listed in alphabetical order. All status codes are severe errors, unless otherwise indicated.

Usage Restrictions:

This section notes any user privileges that are required, and any special conditions that must be established prior to calling this procedure.

Notes:

This optional section contains a detailed usage description of the procedure, as well as references to related information.

CHAPTER 4

SELECTED BASEWAY SUBROUTINES

4.1 Overview

This chapter contains descriptions of various BASEWAY subroutines that are useful in supporting interprocess messages and the reading and writing of records.

NOTE: Subroutines specific to programmable device access are denoted by the letters PDA.

4.2 BSL$ACCESS_DEVICE - Access a Programmable Device (PDA)

BSL$ACCESS_DEVICE

The Access Programmable Device procedure provides a means of establishing an access path to a specified programmable device and setting a default programmable device name. No data is returned by this call.

If a programmable device name is not specified, then the last used programmable device name is used. If a handle is not specified, a default handle is used.

Note that an implicit BSL$ACCESS call is performed by all of the other BSL routines.

Calling Format:

BSL$ACCESS_DEVICE ([handle], [device])

Arguments:

handle.

Optional address of a longword to be used as an internal context identifier. A programmer may choose to keep several handles, each with a different context. This parameter must be initialized to 0 before being specified for the first time in a call, and not modified thereafter. If this parameter is not specified, the system will supply a default context identifier.

device

Optional address of a character string descriptor pointing to the text string naming the programmable device. This programmable device name string must correspond exactly to the name of the device requested. If this parameter is not specified, the last referenced programmable device is used.

Return Status:

L$_NORMAL

Service successfully completed.

BSL$_NOSYSTEM

The programmable device does not have a valid device set associated with it.

BSL$_PDEVDEFLT

The caller did not specify a programmable device name, and did not have a default name set.

BSL$_NOPDEV

The programmable device that was specified does not exist.

4.3 BSL$ACCESS_PORT - Access Another Port

BSL$ACCESS_PORT

The Access Port procedure will find the Port value for a particular named port currently running on an application, device set, or gateway.

This port value directly and uniquely identifies a message port somewhere in the system.

Calling Format:

```
BSL$ACCESS_PORT ( port, [system], port_name, )
```

Arguments:

port

Address of a longword to receive the port value. Port values are 32-bit values that uniquely identify the message port.

system

Optional address of a descriptor pointing to a character string that contains the name of the application, device set, or gateway. If this parameter is not specified, the name of the current application is used.

port_name

Address of a character string descriptor pointing to the text string containing a valid message port name. These names are unique to an application or gateway.

Return Status:

L$_NOBSL

A BASEWAY application is not currently running in this UIC group.

BSL$_NORMAL

Service successfully completed.

SS$_INSFARG

Not enough arguments were passed to the routine.

BSL$_NOBSL

No BASEWAY application is running.

BSL$_NOPORT

If the port is local to the current application, then the specified port name does not exist. If the port is remote, then the problem may be with the Network Interface port.

BSL$_NOSYSTEM

The application, device set, or gateway does not exist.

4.4 BSL$ALLOCATE_DEVICE - Allocate a Programmable Device (PDA)

BSL$ALLOCATE_DEVICE

The Allocate Programmable Device procedure provides a means of requesting exclusive access to a programmable device. This call is often used prior to downline load a device and during complicated diagnostic functions, where it is imperative that no other process access the device. If a programmable device name is specified, then the current default programmable device name is set to the specified name on completion of this call.

Calling Format:

```
BSL$ALLOCATE_DEVICE ( [handle], [device], [flags] )
```

Arguments:

handle

Optional address of a longword to be used as an internal context identifier. A programmer may choose to keep several handles, each with a different context. This parameter must be initialized to 0 before being specified for the first time in a call, and not modified thereafter. If this parameter is not specified, the system will supply a default context identifier.

device

Optional address of a character string descriptor pointing to the text string naming the programmable device. This programmable device name string must correspond exactly to the name of the device requested. If this parameter is not specified, the last referenced programmable device is used.

flags

Optional address of a word containing flags. If the first bit is set in this flag word, then any automatic data collection (polling) of this device is stopped while the device is allocated. If clear, then data collection continues but all other access is denied.

Return Status:

BSL$_NOBSL

A BASEWAY application is not currently running in this UIC group.

BSL$_NORMAL

Service successfully completed.

BSL$_NOSYSTEM

The programmable device does not have a valid device set associated with it.

BSL$_OWNATTACHED

The programmable device that was specified is currently allocated by this process's parent process, and therefore, can be considered allocated to this process.

BSL$_PDEVDEFLT

The caller did not specify a programmable device name, and did not have a default name set.

BSL$_NOPDEV

The programmable device that was specified does not exist.

BSL$_ATTACHED

The programmable device is currently allocated to another process.

4.5 BSL$COMPARE_DEVICE - Compare Programmable Device Logic (PDA)

BSL$COMPARE_DEVICE

The Compare Device Logic procedure will compare two programmable device logic files and report any differences to the caller.

Not all programmable devices support the compare function.

Calling Format:

BSL$COMPARE__DEVICE ([handle], [device], filename)

Arguments:

handle

Optional address of a longword to be used as an internal context identifier. A programmer may choose to keep several handles, each with a different context. This parameter must be initialized to 0 before being specified for the first time in a call, and not modified thereafter. If this parameter is not specified, the system will supply a default context identifier.

device

Optional address of a character string descriptor pointing to the text string naming the programmable device. This programmable device name string must correspond exactly to the name of the device requested. If this parameter is not specified, the last referenced programmable device is used.

filename

Address of a character string descriptor pointing to a VAX/VMS file containing a programmable device logic dump for a similar programmable device.

Return Status:

L$_COMPFAIL

The compare failed due to one or more differences in the logic file and the programmable device's memory.

BSL$_NOBSL

A BASEWAY application is not currently running in this UIC group.

BSL$_NORMAL

Service successfully completed.

BSL$_NOSYSTEM

The programmable device does not have a valid device set associated with it.

BSL$_NOTSUPPORTED

The programmable device does not support this function.

BSL$_PDEVDEFLT

The caller did not specify a programmable device name, and did not have a default name set.

BSL$_NOPDEV

The programmable device that was specified does not exist.

BSL$_ATTACHED

The programmable device is currently allocated to another process.

4.6 BSL$COMPILE_DEVICE_ADDRESS - Precompile an Address (PDA)

BSL$COMPILE_DEVICE_ADDRESS

The Compile Address procedure allows a caller to translate a character string containing a valid programmable device address into an internal format. This internal format may be substituted for the character string device address in other BSL calls.

Precompiling addresses that are to be used over and over can save a substantial amount of overhead, as the address string does not have to be parsed each time.

Calling Format:

```
BSL$COMPILE_DEVICE_ADDRESS ( [handle], [device], address,
                             compiled )
```

Arguments:

handle

Optional address of a longword to be used as an internal context identifier. A programmer may choose to keep several handles, each with a different context. This parameter must be initialized to 0 before being specified for the first time in a call, and not modified thereafter. If this parameter is not specified, the system will supply a default context identifier.

device

Optional address of a character string descriptor pointing to the text string naming the programmable device. This programmable device name string must correspond exactly to the name of the device requested. If this parameter is not specified, the last referenced programmable device is used.

address

Address of a character string descriptor pointing to the text string containing a valid programmable device address.

compiled

Address of a descriptor pointing to a character string to receive the compiled address. The character string must be at least 33 bytes long.

Return Status:

BSL$_NORMAL

Service successfully completed.

BSL$_NOSYSTEM

The programmable device does not have a valid device set associated with it.

BSL$_NOTSUPPORTED

The programmable device does not support this function.

BSL$_PDEVDEFLT

The caller did not specify a programmable device name, and did not have a default name set.

BSL$_NOPDEV

The programmable device that was specified does not exist.

BSL$_BADADDR

The address specified in the "ADDRESS" parameter has an illegal syntax or is outside of the range of valid addresses for this programmable device.

4.7 BSL$CREATE_MESSAGE - Create an Interprocess Message

BSL$CREATE_MESSAGE

The Create Message procedure allocates space for an interprocess message, optionally sets some default message attributes, and returns a pointer to the message.

Messages that are created using this procedure should only be deleted with the BSL$DELETE_MESSAGE procedure.

Calling Format:

```
BSL$CREATE_MESSAGE ( pointer, [size], [code],
                     [dest_port], [source_port] )
```

Arguments:

pointer

Address of a longword to receive the address of the data portion of the message.

size

Optional address of a word containing the size of the data portion of the allocated message. Valid values are in the range of 0 to 8000 bytes. If no size parameter is specified, then the default maximum size of 8000 bytes is used.

code

Optional address of a word containing the message code to be assigned to this message. If no code is specified, then a code must be specified in the BSL$SEND_MESSAGE procedure call.

dest_port

Optional address of a longword containing the port value of the port that this message is to be sent to. If no dest_port is specified, then a destination port must be specified in the BSL$SEND_MESSAGE procedure call.

source_port

Optional address of a longword containing the port value of the port that this message is to be sent from.

Return Status:

BSL$_NORMAL

Service successfully completed.

SS$_INSFARG

Not enough arguments were passed to the routine.

BSL$_BADMSGSIZE

The interprocess message size that was specified is outside of the range of 0 to 8000 bytes.

4.8 BSL$CREATE_NAMED_PORT - Create a Permanent Message Port

BSL$CREATE_NAMED_PORT

The Create Named Port procedure allocates a permanent message port, associates an applicationwide name with it, and assigns it to the calling process. Interprocess messages may then be read from and written to this port.

Message ports created with this procedure should be deleted via the BSL$DELETE_PORT procedure.

Calling Format:

```
BSL$CREATE_NAMED_PORT ( port, name, [size], [queued],
                        [id], )
```

Arguments:

port

Address of a longword to receive the port value. Port values are 32-bit values that uniquely identify the message port.

name

Address of a character string descriptor pointing to a text string containing the name to associate with this port. Names must be alphanumeric symbols, and can be no more than 32 characters in length.

size

Optional address of a word containing the maximum size of a message that can be received through this port. Valid values are in the range of 0 to 8000 bytes. If no size parameter is specified, then the default maximum size of 8000 bytes is used.

queued

Optional address of a word containing the maximum number of messages that can be queued to this port at any one time. If no maximum is specified, then a default value of 10 is assumed.

id

Optional address of a word containing a numeric identifier for this port. Ports that are to receive polled data messages from a gateway must be assigned a unique ID number by the system manager, and use this number each time this port name is created.

Return Status:

BSL$_NOBSL

A BASEWAY application is not currently running in this UIC group.

BSL$_NORMAL

Service successfully completed.

SS$_INSFARG

Not enough arguments were passed to the routine.

L$_BADMSGSIZE

The interprocess message size that was specified is outside of the range of 0 to 8000 bytes.

BSL$_NOBSL

No BASEWAY application is running.

BSL$_PORTEXISTS

A port with this name or ID already exists.

BSL$_TOOMANYPORTS

More than 127 named ports are currently in use by this application.

4.9 BSL$CREATE_PORT - Create a Temporary Message Port

BSL$CREATE_PORT

The Create Port procedure allocates a temporary message port and assigns it to the calling process. Interprocess messages may then be read from and written to this port.

Message ports created with this procedure should be deleted via the BSL$DELETE_PORT procedure.

Calling Format:

```
BSL$CREATE_PORT ( port, [size], [queued], )
```

Arguments:

port

Address of a longword to receive the port value. Port values are 32-bit values that uniquely identify the message port.

size

Optional address of a word containing the maximum size of a message that can be received through this port. Valid values are in the range of 0 to 8000 bytes. If no size parameter is specified, then the default maximum size of 8000 bytes is used.

queued

Optional address of a word containing the maximum number of messages that can be queued to this port at any one time. If no maximum is specified, then a default value of 10 is assumed.

Return Status:

L$_NOBSL

A BASEWAY application is not currently running in this UIC group.

BSL$_NORMAL

Service successfully completed.

SS$_INSFARG

Not enough arguments were passed to the routine.

BSL$_BADMSGSIZE

The interprocess message size that was specified is outside of the range of 0 to 8000 bytes.

BSL$_NOBSL

No BASEWAY application is running.

BSL$_NONAU

More than 127 temporary ports are currently in use by this application.

4.10 BSL$CVT_MX_DX - General Data Type Conversion Routine

BSL$CVT_MX_DX

The Convert Message Descriptor to Data Descriptor procedure converts a data item described by a BASEWAY message descriptor to a VAX standard data descriptor.

Calling Format:

```
BSL$CVT_MX_DX ( src_desc, dest_desc, dest_len )
```

Arguments:

src_desc

Address of a BASEWAY message descriptor. These descriptors are used to identify pieces of data being sent in interprocess messages.

dest_desc

Address of a VAX descriptor pointing to a piece of data. Any data conversions necessary to convert the source data into the destination data are performed, and the resulting data is copied to the memory pointed to by the destination descriptor.

dest_len

Address of a word to receive the length of the data item. This returned from the dest_desc parameter for convenience.

Return Status:

- _$_NORMAL

  Service successfully completed.

4.11 BSL$DATA_TYPE - Find Data Type for a Programmable Device Address (PDA)

BSL$DATA_TYPE

Programmable devices contain a wide variety of data formats and data sizes. The Data Type procedure checks the validity of a programmable device address and returns the physical data type found at this address.

Calling Format:

```
BSL$DATA_TYPE ( [handle], [device], address, type )
```

Arguments:

handle

Optional address of a longword to be used as an internal context identifier. A programmer may choose to keep several handles, each with a different context. This parameter must be initialized to 0 before being specified for the first time in a call, and not modified thereafter. If this parameter is not specified, the system will supply a default context identifier.

device

Optional address of a character string descriptor pointing to the text string naming the programmable device. This programmable device name string must correspond exactly to the name of the device requested. If this parameter is not specified, the last referenced programmable device is used.

address

Address of a character string descriptor pointing to the text string containing a valid programmable device address.

type

Address of a word to receive the type of data referred to by the specified address:

BSL$K_TYPE_BIT	Bit data
BSL$K_TYPE_BYTE	Byte data
BSL$K_TYPE_WORD	Word data
BSL$K_TYPE_LONG	Longword data

Return Status:

BSL$_NORMAL

Service successfully completed.

BSL$_NOSYSTEM

The programmable device does not have a valid device set associated with it.

BSL$_NOTSUPPORTED

The programmable device does not support this function.

BSL$_PDEVDEFLT

The caller did not specify a programmable device name, and did not have a default name set.

BSL$_NOPDEV

The programmable device that was specified does not exist.

BSL$_BADADDR

The address specified in the "ADDRESS" parameter has an illegal syntax or is outside of the range of valid addresses for this programmable device.

4.12 BSL$DEACCESS_DEVICE - Deaccess a Programmable Device (PDA)

BSL$DEACCESS_DEVICE

The Deaccess Programmable Device procedure provides a means of disassociating an access path with a specified programmable device, and clearing the default programmable device name. No data is returned by this call.

Calling Format:

```
BSL$DEACCESS__DEVICE ( [handle] )
```

Arguments:

handle

Optional address of a longword to be used as an internal context identifier. A programmer may choose to keep several handles, each with a different context. This parameter must be initialized to 0 before being specified for the first time in a call, and not modified thereafter. If this parameter is not specified, the system will supply a default context identifier.

Return Status:

BSL$_NORMAL

Service successfully completed.

BSL$_PDEVDEFLT

The caller did not specify a programmable device name, and did not have a default name set.

BSL$_NOPDEV

The programmable device that was specified does not exist.

4.13 BSL$DEALLOCATE_DEVICE - Deallocate a Programmable Device (PDA)

BSL$DEALLOCATE_DEVICE

The Deallocate Programmable Device procedure provides a means of releasing an allocated programmable device so that other users can access it, and so that automatic data collection can resume. If a programmable device name is specified, then the current default programmable device name is set to the specified name on completion of this call.

Calling Format:

BSL$DEALLOCATE ([handle], [device])

Arguments:

handle

Optional address of a longword to be used as an internal context identifier. A programmer may choose to keep several handles, each with a different context. This parameter must be initialized to 0 before being specified for the first time in a call, and not modified thereafter. If this parameter is not specified, the system will supply a default context identifier.

device

Optional address of a character string descriptor pointing to the text string naming the programmable device. This programmable device name string must correspond exactly to the name of the device requested. If this parameter is not specified, the last referenced programmable device is used.

Return Status:

BSL$_NOBSL

A BASEWAY application is not currently running in this UIC group.

BSL$_NORMAL

Service successfully completed.

BSL$_NOSYSTEM

The programmable device does not have a valid device set associated with it.

BSL$_OWNATTACHED

The programmable device that was specified is currently allocated by this process's parent process, and therefore, is not deallocated.

BSL$_NOTATTACHED

The programmable device that was specified in the deallocate operation is not currently allocated by the calling process.

BSL$_PDEVDEFLT

The caller did not specify a programmable device name, and did not have a default name set.

BSL$_NOPDEV

The programmable device that was specified does not exist.

4.14 BSL$DELETE_MESSAGE - Delete an Interprocess Message

BSL$DELETE_MESSAGE

The Delete Message procedure deallocates space that an interprocess message occupies.

Calling Format:

```
BSL$DELETE_MESSAGE ( pointer )
```

Arguments:

pointer

Address of a longword to receive the address of the data portion of the message.

Return Status:

BSL$_NORMAL

Service successfully completed.

4.15 BSL$DELETE_PORT - Delete a Message Port

BSL$DELETE_PORT

The Delete Port procedure frees up a message port that was assigned to the calling process.

Calling Format:

```
BSL$DELETE_PORT ( port )
```

Arguments:

port

Address of a longword to receive the port value. Port values are 32-bit values that uniquely identify the message port.

Return Status:

BSL$_NOBSL

A BASEWAY application is not currently running in this UIC group.

BSL$_NORMAL

Service successfully completed.

BS$_INSFARG

Not enough arguments were passed to the routine.

BSL$_NOPORT

The specified port does not exist, or is not assigned to the calling process.

4.16 BSL$DOWNLOAD_DEVICE - Download Logic to a Programmable Device (PDA)

BSL$DOWNLOAD_DEVICE

The Download Programmable Device procedure provides a means of loading a VAX/VMS file containing device logic into a programmable device.

Not all programmable devices support this function.

Calling Format:

BSL$DOWNLOAD__DEVICE ([handle], [device], filename)

Arguments:

handle

Optional address of a longword to be used as an internal context identifier. A programmer may choose to keep several handles, each with a different context. This parameter must be initialized to 0 before being specified for the first time in a call, and not modified thereafter. If this parameter is not specified, the system will supply a default context identifier.

device

Optional address of a character string descriptor pointing to the text string naming the programmable device. This programmable device name string must correspond exactly to the name of the device requested. If this parameter is not specified, the last referenced programmable device is used.

filename

Address of a character string descriptor containing a VAX/VMS filename of a file that contains programmable device logic for a similar device.

Return Status:

BSL$_NOBSL

A BASEWAY application is not currently running in this UIC group.

BSL$_NORMAL

Service successfully completed.

BSL$_NOSYSTEM

The programmable device does not have a valid device set associated with it.

BSL$_NOTSUPPORTED

The programmable device does not support this function.

BSL$_PDEVDEFLT

The caller did not specify a programmable device name, and did not have a default name set.

BSL$_NOPDEV

The programmable device that was specified does not exist.

BSL$_ATTACHED

The programmable device is currently allocated to another process.

4.17 BSL$GET_DATA_INFO - Get Data Information

BSL$GET_DATA_INFO

The Get Data Information procedure provides information about a piece of data known to BASEWAY. Information available includes data type, name, internal identifier, etc.

Calling Format:

```
BSL$GET_DATA_INFO ( [id], [name], item_list )
```

Arguments:

id

Optional address of a longword containing the data identifier value.

name

Optional address of a character string descriptor pointing to the name of this piece of data. Either the ID or the NAME parameter must be specified.

item_list

Address of a list of item descriptors that describe specific attributes that are requested, and point to the buffers to receive the information.

Return Status:

BSL$_NORMAL

Service successfully completed.

BSL$_NODATA

The data point specified by either a name or an ID does not exist.

BSL$_INVCALL

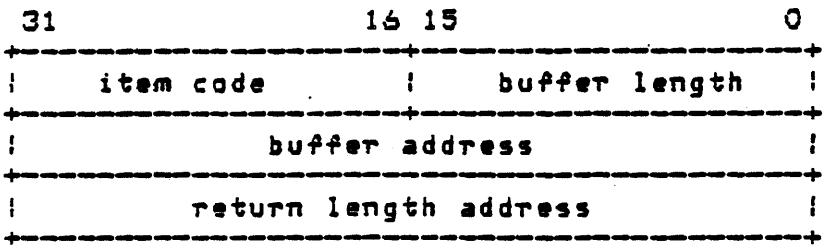
Neither the name or ID was specified

BSL$_TRUNC

User-specified buffer to receive the data was too small.

Notes:

The item descriptors in the item list have the following format:



The format of the item list is described in Section 2.3.3 of the VAX/VMS System Services Reference Manual. The item codes for the procedure call are described in the table below.

Item Identifier	Data Type	Information Specified	
BSL$K_DI_DID	value	Data point internal identifier	
BSL$K_DI_NAME	string	Data point name	
BSL$K_DI_FLAGS	value	data point status flags	
		Symbol	Meaning
		BSL$M_DI_STATUS	status point
		BSL$M_DI_VALUE	value point
		BSL$M_DI_STRUCTURE	structure of points
		BSL$M_DI_GROUP	point group

4.18 BSL$GET_DEVICE_ATTRIBUTES - Get Device Attributes (PDA)

BSL$GET_DEVICE_ATTRIBUTES

The Get Attributes procedure provides allocation, definition, and status information about a programmable device. This call may be used to determine whether a device is in a proper state for downline loading and to write device status reports and utilities. If a programmable device name is specified, then the current default programmable device name is set to the specified name on completion of this call.

Calling Format:

BSL$GET_DEVICE_ATTRIBUTES ([handle], [device], item_list)

Arguments:

handle

Optional address of a longword to be used as an internal context identifier. A programmer may choose to keep several handles, each with a different context. This parameter must be initialized to 0 before being specified for the first time in a call, and not modified thereafter. If this parameter is not specified, the system will supply a default context identifier.

device

Optional address of a character string descriptor pointing to the text string naming the programmable device. This programmable device name string must correspond exactly to the name of the device requested. If this parameter is not specified, the last referenced programmable device is used.

item_list

Address of a list of item descriptors that describe specific attributes that are requested, and point to the buffers to receive the information.

Return Status:

BSL$_NOBSL

A BASEWAY application is not currently running in this UIC group.

BSL$_NORMAL

Service successfully completed.

BSL$_NOSYSTEM

The programmable device does not have a valid device set associated with it.

BSL$_PDEVDEFLT

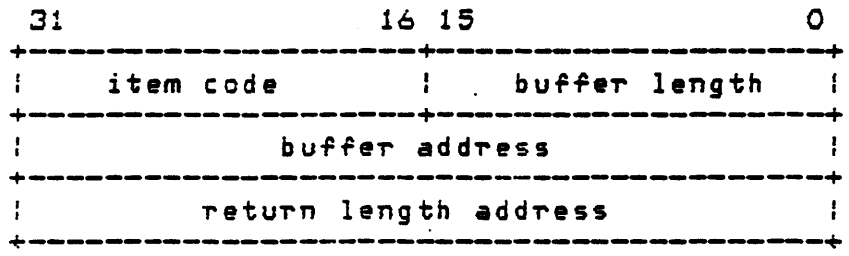
The caller did not specify a programmable device name and did not have a default name set.

BSL$_NOPDEV

The programmable device that was specified does not exist.

Notes:

The item descriptors in the item list have the following format:



The format of the item list is described in Section 2.3.3 of the VAX/VMS System Services Reference Manual. The item codes for the procedure call are described in the table below.

Item Identifier	Data Type	Information Specified
BSL$K_DA_ID	value	Programmable device internal ID number
BSL$K_DA_NAME	string	Programmable device name.
BSL$K_DA_FLAGS	value	Programmable device status flags.
		Symbol / **Meaning**
		BSL$M_DA_WRITEPROT — Write protected
		BSL$M_DA_ATTACHED — Allocated to process
		BSL$M_DA_REACHABLE — Device is reachable
		BSL$M_DA_DISABLED — Device is disabled
		BSL$M_DA_OFFLINE — Device is offline
		BSL$M_DA_POLLED — Polled device
		BSL$M_DA_ADDRESSABLE — Has addressable memory locations
		BSL$M_DA_INTERFACE — Interface device
		BSL$M_DA_READ_ONLY — Data is read-only
		BSL$M_DA_PRODUCTION — Production device
		BSL$M_DA_BAR_CODE — Bar code device
BSL$K_DA_INTERFACE	string	SHOP FLOOR GATEWAY device specification for communications interface to this device
BSL$K_DA_STATION	value	Station number for this device (if it is a multidrop communications protocol)
BSL$K_DA_MFR_NAME	string	Manufacturer of this device
BSL$K_DA_MODEL_NAME	string	Model number of this device
BSL$K_DA_MEMSIZE	value	Memory size of this device
BSL$K_DA_DELAY	value	Timeout delay factor (in seconds)

BSL$_K_DA_LOCATION	string	Location description string
BSL$K_DA_DESCRIPTION	string	Device description string
BSL$K_DA_CONTRACTOR	string	Contractor description string
BSL$K_DA_NET_NAME	string	Communications Network name
BSL$K_DA_DEVICE_SET	string	Device set associated with this device
BSL$K_DA_GATEWAY	string	Gateway currently communicating with this device

4.19 BSL$GET_SESSION_INFO - Retrieve Current Session Information

BSL$GET_SESSION_INFO

The Get Session Information procedure provides information about the current terminal session. Information available includes current privileges, default menu forms, default printers, etc.

This procedure may only be called in the context of an interactive process.

Calling Format:

```
BSL$GET_SESSION_INFO ( item_list )
```

Arguments:

item_list

Address of a list of item descriptors that describe specific attributes that are requested, and point to the buffers to receive the information.

Return Status:

BSL$_NOBSL

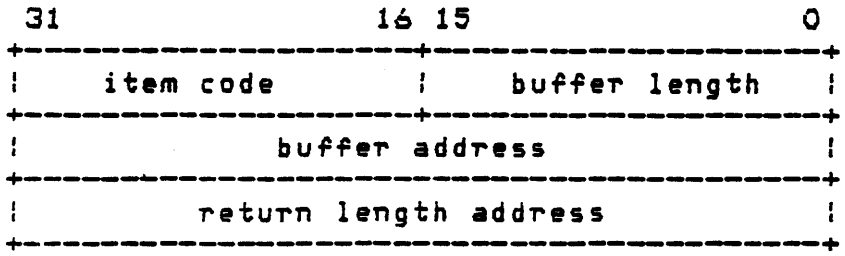
A BASEWAY application is not currently running in this UIC group.

BSL$_NORMAL

Service successfully completed.

Notes:

The item descriptors in the item list have the following format:



The format of the item list is described in Section 2.3.3 of the VAX/VMS System Services Reference Manual. The item codes for the procedure call are described in the table below.

Item Identifier	Data Type	Information Specified
BSLK_SI_TERMINAL	string	Terminal device name
BSL$K_SI_USERNAME	string	VAX/VMS user name
BSL$K_SI_NAME	string	User's name
BSL$K_SI_NICKNAME	string	User's nickname
BSL$K_SI_ADDR	string	User's address
BSL$K_SI_PHONE	string	User's phone number
BSL$K_SI_TITLE	string	User's title or position
BSL$K_SI_DEPARTMENT	string	User's department name
BSL$K_SI_PRINTER	string	Default printer for this session
BSL$K_SI_PRIVS	string	Privilege mask for this session
BSL$K_SI_CUST_PRIVS	string	Customer-defined privileges
BSL$K_SI_LOCATION	string	Terminal location description

4.20 BSL$GET_SYSTEM_INFO - Get System Attributes

BSL$GET_SYSTEM_INFO

The Get System Information procedure provides definition and status information about an application, device set, or a gateway.

The system may be selected by specifying either a NAX or a system name. If neither are specified, then the current application's attributes are returned. If a system name of '*' is specified, then a wildcard lookup is performed, and each successive call returns another system until the status code BSL$_NOSYSTEM is returned.

Calling Format:

```
BSL$GET_SYSTEM_INFO ( [nax], [name], item_list)
```

Arguments:

nax

Optional address of a word that contains a system internal identifier.

name

Optional address of a character string descriptor pointing to the text string naming the application, device set, or gateway.

item_list

Address of a list of item descriptors that describe specific attributes that are requested, and point to the buffers to receive the information.

Return Status:

BSL$_NOBSL

A BASEWAY application is not currently running in this UIC group.

BSL$_NORMAL

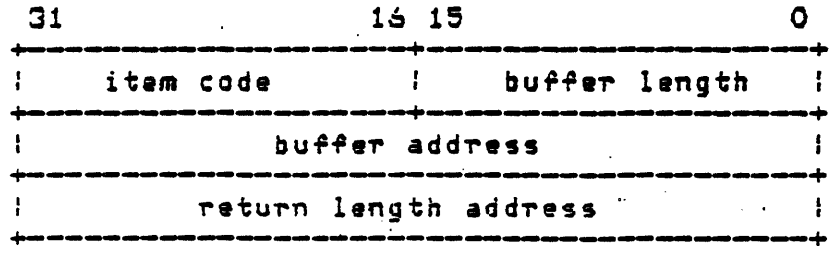
Service successfully completed.

BSL$_NOSYSTEM

The specified system does not exist, or there are no more systems if this was a wildcard lookup.

Notes:

The item descriptors in the item list have the following format:



The format of the item list is described in Section 2.3.3 of the VAX/VMS System Services Reference Manual. The item codes for the procedure call are described in the table below.

Item Identifier	Data Type	Information Specified
BSL$K_AI_ID	value	System internal identifier number
BSL$K_AI_NAME	string	System name
BSL$K_AI_FLAGS	value	System status flags
		Symbol: BSL$M_AI_APPLICATION — Meaning: Application
		Symbol: BSL$M_AI_DEVICESET — Meaning: Device Set
		Symbol: BSL$M_AI_GATEWAY — Meaning: SHOP FLOOR GATEWAY system
		Symbol: BSL$M_AI_REACHABLE — Meaning: Reachable
BSL$K_AI_DESCRIPTION	string	system description

4.21 BSL$LOG_EVENT - Log a System Event in History File

BSL$LOG_EVENT

The Log Event procedure is a user-callable routine to write an entry to the system audit facility. This procedure accepts a formatted ASCII text string from the caller and sends a message to the EVENT_LOG process. Depending on specified flags, the text string may be logged in the System Audit file, written to OPER terminals, or both.

Calling Format:

```
BSL$LOG_EVENT ( code, flags, text )
```

Arguments:

code

Address of a longword containing a bit mask of event codes for this event. The following event codes may be present in any combination:

```
BSL$M_ET_SYSTEM  - System-related event
BSL$M_ET_CONFIG  - Configuration event
BSL$M_ET_NETWORK - Network event
BSL$M_ET_GATEWAY - SHOP FLOOR GATEWAY event
BSL$M_ET_PD      - Programmable Device event
BSL$M_ET_USER    - User-related event
BSL$M_ET_DATA    - Data exception event
BSL$M_ET_APPL    - Application-specific event
```

flags

Address of a longword containing flags for this event. The following flags may be present:

```
BSL$M_EF_FILE - Audit event in file
BSL$M_EF_OPER - Print event on OPER terminals
```

text

Address of a descriptor pointing to a character string containing an event description. This character string may be up to 120 bytes in length, and may contain any printable characters. If multiple lines are desired, embedded carriage return/linefeeds may be included.

Return Status:

BSL$_NOBSL

A BASEWAY application is not currently running in this UIC group.

BSL$_NORMAL

Normal successful completion.

BSL$_NOPORT

The Event Logger message port (EVENT_LOG) does not exist in the current application.

4.22 BSL$READ_DEVICE_DATA - Read Data from a Programmable Device (PDA)

BSL$READ_DEVICE_DATA

The Read Data procedure allows a caller to perform a logical data read from any address of a programmable device. If a programmable device name is specified, then the current default programmable device name is set to the specified name on completion of this call.

Calling Format:

```
BSL$READ_DEVICE_DATA ( [handle], [device], address, count,
                       buffer )
```

Arguments:

handle

Optional address of a longword to be used as an internal context identifier. A programmer may choose to keep several handles, each with a different context. This parameter must be initialized to 0 before being specified for the first time in a call, and not modified thereafter. If this parameter is not specified, the system will supply a default context identifier.

device

Optional address of a character string descriptor pointing to the text string naming the programmable device. This programmable device name string must correspond exactly to the name of the device requested. If this parameter is not specified, the last referenced programmable device is used.

address

Address of a character string descriptor pointing to the text string containing a valid programmable device address.

count

Address of a word that contains the amount of data to be read. For example, if the address specified refers to individual bits, or coils, then the count is the number of bits. If the address specified refers to a 16-bit word, then the count is the number of words. Upon completion of this call, this parameter is updated to reflect the amount of data that is actually returned in the buffer.

buffer

Address of a buffer where the programmable device data should be stored. It is up to the caller to insure that this buffer is large enough to contain the resulting data.

Return Status:

BSL$_NOBSL

A BASEWAY application is not currently running in this UIC group.

BSL$_NORMAL

Service successfully completed.

BSL$_NOSYSTEM

The programmable device does not have a valid device set associated with it.

BSL$_NOTSUPPORTED

The programmable device does not support this function.

BSL$_PDEVDEFLT

The caller did not specify a programmable device name and did not have a default name set.

BSL$_NOPDEV

The programmable device that was specified does not exist.

BSL$_ATTACHED

The programmable device is currently allocated to another process.

BSL$_BADADDR

The address specified in the "ADDRESS" parameter has an illegal syntax, or is outside of the range of valid addresses for this programmable device.

4.23 BSL$READ_DEVICE_STATUS - Read Status Info from a Programmable Device (PDA)

BSL$READ_DEVICE_STATUS

The Read Status procedure allows a caller to get the status information from a programmable device. The format of the status buffer is device specific, and may contain memory sizes, key switch locations, etc.

If a programmable device name is specified, then the current default programmable device name is set to the specified name on completion of this call.

Calling Format:

BSL$READ_STATUS ([handle], [device], count, buffer,)

Arguments:

handle

Optional address of a longword to be used as an internal context identifier. A programmer may choose to keep several handles, each with a different context. This parameter must be initialized to 0 before being specified for the first time in a call, and not modified thereafter. If this parameter is not specified, the system will supply a default context identifier.

device

Optional address of a character string descriptor pointing to the text string naming the programmable device. This programmable device name string must correspond exactly to the name of the device requested. If this parameter is not specified, the last referenced programmable device is used.

count

Address of a word to receive the number of bytes of status information read.

buffer

Address of a buffer where the programmable device status data should be stored. It is up to the caller to insure that this buffer is large enough to contain the resulting data.

Return Status:

BSL$_NOBSL

A BASEWAY application is not currently running in this UIC group.

BSL$_NORMAL

Service successfully completed.

BSL$_NOSYSTEM

The programmable device does not have a valid device set associated with it.

BSL$_NOTSUPPORTED

The programmable device does not support this function.

BSL$_PDEVDEFLT

The caller did not specify a programmable device name, and did not have a default name set.

BSL$_NOPDEV

The programmable device that was specified does not exist.

BSL$_ATTACHED

The programmable device is currently allocated to another process.

4.24 BSL$RECEIVE_MESSAGE - Read a Message from a Port

BSL$RECEIVE_MESSAGE

The Receive Message procedure will read an interprocess message from any sender. The calling process must have already established ownership of the port (via BSL$CREATE_PORT or BSL$CREATE_NAMED_PORT).

If no messages are currently queued, this routine will wait until one is received or until the timeout value (if specified) expires.

Calling Format:

```
BSL$RECEIVE_MESSAGE ( port, [pointer], [buffer],
                      [src_port], [code], [size],
                      [timeout] )
```

Arguments:

port

Address of a longword to containing the port value of the port to receive the message.

pointer

Optional address of a longword to receive the address of the message received. This parameter may not be specified if the buffer parameter is specified.

buffer

Optional address of a buffer to receive the message data. If this parameter is specified, then the pointer parameter may not be specified.

source_port

Optional address of a longword to receive the port value for the port that sent this message.

code

Optional address of a word to receive the message code for this message.

size

Optional address of a word to receive the size of this message.

timeout

Optional address of a quadword containing a VMS system delta time to wait before returning if no messages are outstanding.

Return Status:

BSL$_NOBSL

A BASEWAY application is not currently running in this UIC group.

BSL$_NORMAL

Service successfully completed.

SS$_INSFARG

Not enough arguments were passed to the routine.

BSL$_BADMSGSIZE

The interprocess message size that was specified is outside of the range of 0 to 8000 bytes.

BSL$_NOBSL

No BASEWAY application is running.

4.25 BSL$SEND_MESSAGE - Send a Message to a Port

BSL$SEND_MESSAGE

The Send Message procedure will send an interprocess message to any port in the system. The destination port value must have already been found by calling the BSL$ACCESS_PORT procedure.

The Send Message procedure may be passed either a pointer to a message (created with BSL$CREATE_MESSAGE or BSL$RECEIVE_MESSAGE), or a buffer containing data to be sent.

Calling Format:

```
BSL$SEND_MESSAGE ( [source_port], [dest_port],
                   [pointer], [buffer], [code],
                   [size] )
```

Arguments:

source_port

Optional address of a longword to containing a port value for a port that a response should be returned to. If no response is requested, then no source port parameter need be supplied.

dest_port

Optional address of a longword to containing the port value of the port that this message is to be sent to. This parameter is required unless a pointer to a message that already contains a destination port is passed.

pointer

Optional address of a longword that contains the address of a message to be sent. This parameter may not be specified if the buffer parameter is specified.

buffer

Optional address of a buffer of data to be sent. If this parameter is specified, then the code, dest_port, and size parameters are mandatory.

code

Optional address of a word that contains the message code for this message.

size

Optional address of a word containing the size of this message.

Return Status:

BSL$_NOBSL

A BASEWAY application is not currently running in this UIC group.

BSL$_NORMAL

Service successfully completed.

SS$_INSFARG

Not enough arguments were passed to the routine.

BSL$_BADMSGSIZE

The interprocess message size that was specified is outside of the range of 0 to 8000 bytes.

BSL$_NOBSL

No BASEWAY application is running.

4.26 BSL$SET_DEVICE_ATTRIBUTES - Change Current Device Attributes (PDA)

BSL$SET_DEVICE_ATTRIBUTES

The Set Attributes procedure provides a method of setting definition and status information for a specified programmable device. This call may be used in writing system status utilities. If a programmable device name is specified, then the current default programmable device name is set to the specified name on completion of this call.

Calling Format:

BSL$SET_DEVICE_ATTRIBUTES ([handle], [device], item_list)

Arguments:

handle

Optional address of a longword to be used as an internal context identifier. A programmer may choose to keep several handles, each with a different context. This parameter must be initialized to 0 before being specified for the first time in a call, and not modified thereafter. If this parameter is not specified, the system will supply a default context identifier.

device

Optional address of a character string descriptor pointing to the text string naming the programmable device. This programmable device name string must correspond exactly to the name of the device requested. If this parameter is not specified, the last referenced programmable device is used.

item_list

Address of a list of item descriptors that describe specific attributes that are requested, and point to the buffers to receive the information.

Return Status:

BSL$_NOBSL

A BASEWAY application is not currently running in this UIC group.

BSL$_NORMAL

Service successfully completed.

BSL$_NOSYSTEM

The programmable device does not have a valid device set associated with it.

BSL$_PDEVDEFLT

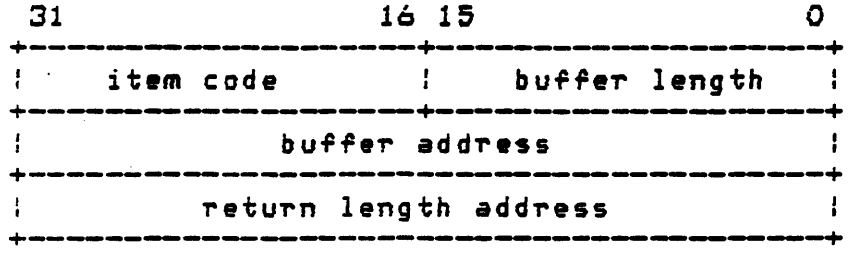
The caller did not specify a programmable device name, and did not have a default name set.

BSL$_NOPDEV

The programmable device that was specified does not exist.

Notes:

The item descriptors in the item list have the following format:



The format of the item list is described in Section 2.3.3 of the VAX/VMS System Services Reference Manual. The item codes for the procedure call are described in the table below.

Item Identifier	Data Type	Information Specified
BSL$K_DA_NAME.	string	Programmable device name.
BSL$K_DA_FLAGS	value	Programmable device status flags.
		Symbol: BSL$M_DA_WRITEPROT — Meaning: Write protected
		Symbol: BSL$M_DA_DISABLED — Meaning: Device is disabled
		Symbol: BSL$M_DA_OFFLINE — Meaning: Device is offline
BSL$K_DA_LOCATION	string	Location description string
BSL$K_DA_DESCRIPTION	string	Device description string
BSL$K_DA_CONTRACTOR	string	Contractor description string

27 BSL$SLEEP - Sleep for Specified Time Interval

BSL$SLEEP

This procedure will suspend the process execution for the specified amount of time. This procedure may be useful for dynamic terminal displays, where the program must pause for n seconds between refreshes.

The specified time interval is expected to be a valid ASCII delta time, as specified in the description of the SYS$BINTIM system service call.

Calling Format:

BSL$SLEEP (ascii_time)

Arguments:

ascii_time

Address of a descriptor pointing to a character string containing a valid VAX/VMS delta time.

Return Status:

BSL$_NORMAL

Service successfully completed.

4.28 BSL$START_DEVICE - Start a Programmable Device (PDA)

BSL$START_DEVICE

The Start Device procedure provides a means of starting a programmable device that is currently stopped. This may be necessary after downline loading a new programmable device program, performing maintenance, etc.

Some programmable devices, such as bar code readers, may not perform this function. In those cases, an error status return of BSL$_NOTSUPPORTED is returned.

Calling Format:

BSL$START_DEVICE ([handle], [device])

Arguments:

handle

Optional address of a longword to be used as an internal context identifier. A programmer may choose to keep several handles, each with a different context. This parameter must be initialized to 0 before being specified for the first time in a call, and not modified thereafter. If this parameter is not specified, the system will supply a default context identifier.

device

Optional address of a character string descriptor pointing to the text string naming the programmable device. This programmable device name string must correspond exactly to the name of the device requested. If this parameter is not specified, the last referenced programmable device is used.

Return Status:

BSL$_NOBSL

A BASEWAY application is not currently running in this UIC group.

BSL$_NORMAL

Service successfully completed.

BSL$_NOSYSTEM

The programmable device does not have a valid device set associated with it.

BSL$_NOTSUPPORTED

The programmable device does not support this function.

BSL$_PDEVDEFLT

The caller did not specify a programmable device name, and did not have a default name set.

BSL$_NOPDEV

The programmable device that was specified does not exist.

BSL$_NOTSUPPORTED

The programmable device that was specified does not support this operation.

BSL$_ATTACHED

The programmable device is currently allocated to another process.

4.29 BSL$STOP_DEVICE - Stop a Programmable Device (PDA)

BSL$STOP_DEVICE

The Stop Device procedure provides a means of stopping a programmable controller. This may be necessary for downline loading a new programmable device program, maintenance, etc.

Some programmable devices, such as bar code readers, may not perform this function. In those cases, an error status return of BSL$_NOTSUPPORTED is returned.

Calling Format:

BSL$STOP_DEVICE ([handle], [device]),

Arguments:

handle

Optional address of a longword to be used as an internal context identifier. A programmer may choose to keep several handles, each with a different context. This parameter must be initialized to 0 before being specified for the first time in a call, and not modified thereafter. If this parameter is not specified, the system will supply a default context identifier.

device

Optional address of a character string descriptor pointing to the text string naming the programmable device. This programmable device name string must correspond exactly to the name of the device requested. If this parameter is not specified, the last referenced programmable device is used.

Return Status:

BSL$_NOBSL

A BASEWAY application is not currently running in this UIC group.

BSL$_NORMAL

Service successfully completed.

BSL$_NOSYSTEM

The programmable device does not have a valid device set associated with it.

BSL$_NOTSUPPORTED

The programmable device does not support this function.

BSL$_PDEVDEFLT

The caller did not specify a programmable device name and did not have a default name set.

BSL$_NOPDEV

The programmable device that was specified does not exist.

BSL$_NOTSUPPORTED

The programmable device that was specified does not support this operation.

BSL$_ATTACHED

The programmable device is currently allocated to another process.

4.30 BSL$UPLOAD_DEVICE - Upload Logic from a Programmable Device (PDA)

BSL$UPLOAD_DEVICE

The Upload Programmable Device procedure provides a means of loading programmable device logic from a device into a VAX/VMS file.

Not all programmable devices support this function.

Calling Format:

BSL$UPLOAD__DEVICE ([handle], [device], filename)

Arguments:

handle

Optional address of a longword to be used as an internal context identifier. A programmer may choose to keep several handles, each with a different context. This parameter must be initialized to 0 before being specified for the first time in a call, and not modified thereafter. If this parameter is not specified, the system will supply a default context identifier.

device

Optional address of a character string descriptor pointing to the text string naming the programmable device. This programmable device name string must correspond exactly to the name of the device requested. If this parameter is not specified, the last referenced programmable device is used.

filename

Address of a character string descriptor containing a VAX/VMS filename of a file that is to contain the programmable device logic for this device.

Return Status:

BSL$_NOBSL

A BASEWAY application is not currently running in this UIC group.

BSL$_NORMAL

Service successfully completed.

BSL$_NOSYSTEM

The programmable device does not have a valid device set associated with it.

BSL$_NOTSUPPORTED

The programmable device does not support this function.

BSL$_PDEVDEFLT

The caller did not specify a programmable device name and did not have a default name set.

BSL$_NOPDEV

The programmable device that was specified does not exist.

BSL$_ATTACHED

The programmable device is currently allocated to another process.

4.31 BSL$WRITE_DEVICE_DATA - Write Data To a Programmable Device (PDA)

BSL$WRITE_DEVICE_DATA

The Write Data procedure allows a caller to write data to any address of a programmable device. If a programmable device name is specified, then the current default programmable device name is set to the specified name on completion of this call.

Calling Format:

```
BSL$WRITE_DEVICE_DATA ( [handle], [device], address, count,
                        buffer )
```

Arguments:

handle

Optional address of a longword to be used as an internal context identifier. A programmer may choose to keep several handles, each with a different context. This parameter must be initialized to 0 before being specified for the first time in a call, and not modified thereafter. If this parameter is not specified, the system will supply a default context identifier.

device

Optional address of a character string descriptor pointing to the text string naming the programmable device. This programmable device name string must correspond exactly to the name of the device requested. If this parameter is not specified, the last referenced programmable device is used.

address

Address of a character string descriptor pointing to the text string containing a valid programmable device address.

count

Address of a word that contains the amount of data to be written. For example, if the address specified refers to individual bits, or coils, then count is the number of bits to write. If the address specified refers to 16-bit words, then count is the number of words to write.

buffer

Address of a buffer where the programmable device data is stored. It is up to the caller to insure that this buffer contains enough data to fulfill the write request.

Return Status:

BSL$_NOBSL

A BASEWAY application is not currently running in this UIC group.

BSL$_NORMAL

Service successfully completed.

BSL$_NOSYSTEM

The programmable device does not have a valid device set associated with it.

BSL$_NOTSUPPORTED

The programmable device does not support this function.

BSL$_PDEVDEFLT

The caller did not specify a programmable device name and did not have a default name set.

BSL$_NOPDEV

The programmable device that was specified does not exist.

BSL$_ATTACHED

The programmable device is currently allocated to another process.

BSL$_BADADDR

The address specified in the "ADDRESS" parameter has an illegal syntax, or is outside of the range of valid addresses for this

programmable device.

4.32 BSL$WRITE_VERIFY_DEVICE_DATA - Write Data To a Programmable

Device and Verify (PDA)

BSL$WRITE_VERIFY_DEVICE_DATA

The Write Verify Data procedure allows a caller to perform write data to any address of a programmable device. If a programmable device name is specified, then the current default programmable device name is set to the specified name on completion of this call.

Calling Format:

```
BSL$WRITE_VERIFY_DATA ( [handle], [device], address,
                        count, buffer )
```

Arguments:

handle

Optional address of a longword to be used as an internal context identifier. A programmer may choose to keep several handles, each with a different context. This parameter must be initialized to 0 before being specified for the first time in a call, and not modified thereafter. If this parameter is not specified, the system will supply a default context identifier.

device

Optional address of a character string descriptor pointing to the text string naming the programmable device. This programmable device name string must correspond exactly to the name of the device requested. If this parameter is not specified, the last referenced programmable device is used.

address

Address of a character string descriptor pointing to the text string containing a valid programmable device address.

count

Address of a word that contains the amount of data to be written. For example, if the address specified refers to individual bits, or coils, then count is the number of bits to write. If the address specified refers to 16-bit words, then count is the number of words to write.

buffer

Address of a buffer where the programmable device data is stored. It is up to the caller to insure that this buffer contains enough data to fulfill the write request.

Return Status:

BSL$_NOBSL

A BASEWAY application is not currently running in this UIC group.

BSL$_NORMAL

Service successfully completed.

BSL$_NOSYSTEM

The programmable device does not have a valid device set associated with it.

BSL$_NOTSUPPORTED

The programmable device does not support this function.

BSL$_PDEVDEFLT

The caller did not specify a programmable device name and did not have a default name set.

BSL$_NOPDEV

The programmable device that was specified does not exist.

BSL$_ATTACHED

The programmable device is currently allocated to another process.

BSL$_BADADDR

The address specified in the "ADDRESS" parameter has an illegal syntax, or is outside of the range of valid addresses for this

programmable device.

APPENDIX A

BASEWAY SUBROUTINE CALLS

A.1 BASEWAY Language-Independent Notation

BASEWAY routines are invoked according to rules specified in the VAX Procedure Calling and Condition Handling Standard (Appendix C of the VAX Run-Time Library Reference Manual). The complete notation for describing VAX calls is documented in Appendix C of the VAX Guide to Creating Modular Library Procedures.

BASEWAY routines can be invoked as subroutines or as functions:

As a subroutine:

```
CALL BSL$xxx (parameter1, parameter2, ... )
```

As a function:

```
VMS_stat.wlc.v = BSL$xxx (parameter1, parameter2, ... )
```

The access type, data type, passing mechanism, and parameter form are described in the following prescribed order:

<parameter-name>.<access type><data type>.<passing mechanism><parameter form>

A.2 Procedure Parameter Notation for BASEWAY Calls

BASEWAY uses a subset of the VAX procedure parameter notation. The following table explains the notation used for access type, data type, passing mechanism, and parameter form.

Notation	<access type>	Comments
m	Modify access	Parameters for both input and output
r	Read-only access	Parameters for input
w	Write-only access	Parameters for output

Notation	<data type>
a	Virtual address
l	Longword integer (signed)
lc	Longword return status
q	Quadword integer (unsigned)
t	Character-coded text string
v	Aligned bit string
w	Word integer (signed)
x	Any data type

Notation	<passing mechanism>	Comments
d	By descriptor	BASEWAY passing mechanism for character strings and integer arrays
r	By reference	BASEWAY passing mechanism for integers

tation	<parameter form>
a	Array reference or descriptor
x1	Fixed-length or dynamic string descriptor

Procedure Parameter Notation

BSL$ACCESS_DEVICE ([handle.ma.r], [name.rt.dx1])

handle	context identifier
name	programmable device name

Places the current context pointer to a specific programmable device, and returns the access status.

BSL$ACCESS_PORT (port.wl.r, [system.rt.dx1], name.rt.dx1,)

port	port value
system	application, device set or gateway name
name	port name

Returns the port value for a particular named port currently active on an application, device set, or gateway.

BSL$ALLOCATE_DEVICE ([handle.ma.r], [name.rt.dx1], [flags.rv.r])

handle	context identifier
name	programmable device name
flags	allocation flags

Marks a programmable device for exclusive use by the calling process. Causes any automatic data gathering (polling) to cease. Usually used when performing a diagnostic function or when downline loading a programmable device.

BSL$COMPARE_DEVICE ([handle.ma.r], [name.rt.dx1], file.rt.dx1)

handle	context identifier
name	programmable device name
file	VAX/VMS file name

Compares two programmable device logic files and returns a summary of differences.

Procedure Parameter Notation

```
BSL$COMPILE_DEVICE_ADDRESS ( [handle.ma.r], [name.rt.dx1], address.rt.dx1,
                        compiled.rt.dx1 )

                handle          context identifier
                name            programmable device name
                address         address in programmable device
                compiled        precompiled address string

                Parses an address for this programmable device, and
                returns the precompiled address string. This string
                is nonprintable, but may be used in other
                calls in place of the address string.

BSL$CREATE_MESSAGE ( pointer.wa.r, [size.rw.r], [code.rw.r],
                     [dest_port.rl.r], [source_port.rl.r] )

                pointer         address of message
                size            size of data buffer in bytes
                code            message code
                dest_port       port value
                source_port     port value

                Allocates space for an interprocess message, and
                returns a pointer to the data area of this message.

BSL$CREATE_NAMED_PORT ( port.wl.r, name.rt.dx1, [size.rw.r],
                        [queued.rw.r], [id.rw.r], )

                port            port value
                name            name given to this port
                size            maximum buffer that can be received
                queued          maximum number of messages that
                                can be queued to this port
                id              identification number (64-127)

                Allocates a permanent message port, associates
                the given name with it, and assigns it to the
                calling process.
```

More efficient; "Logical translation" takes time. (Using RMS file ID faster than using file name.)

Procedure Parameter Notation

```
BSL$CREATE_PORT ( port.wl.r, [size.rw.r], [queued.rw.r], )

                port        port value
                size        maximum buffer that can be received
                queued      maximum number of messages that
                            can be queued to this port

                Allocates a temporary message port and assigns
                it to the calling process.


BSL$CVT_MX_DX ( src_desc.rx.dx1, dest_desc.wx.dx1, dest_len.ww.r )

                src_desc    address of message descriptor
                dest_desc   address of VAX data descriptor
                dest_len    size of resulting value

                Converts data described by src_descriptor into the
                data type described by dest_desc, and copies the
                resulting value to the space described by dest_desc.

BSL$DATA_TYPE ( [handle.ma.r], [name.rt.dx1], address.rt.dx1, type )

                handle      context identifier
                name        programmable device name
                address     starting memory location address
                type        data type at this address

                Parses an address for this programmable device, and
                returns the type code for the data at this address:

                BSL$K_DT_BIT  - 1-bit or coil address
                BSL$K_DT_BYTE - Byte data at this address
                BSL$K_DT_WORD - 16-bit word data
                BSL$K_DT_LONG - 32-bit longword data


BSL$DEACCESS_DEVICE ( [handle.ma.r] )

                handle      context identifier

                Clears any programmable device context previously
                associated with this handle.
```

Procedure Parameter Notation

BSL$DEALLOCATE_DEVICE ([handle.ma.r], [name.rt.dx1])

handle	context identifier
name	programmable device name

Frees a previously allocated programmable device, and causes normal data-gathering functions to resume if possible.

BSL$DELETE_MESSAGE (pointer.ma.r)

pointer	address of message

Releases space occupied by this message.

BSL$DELETE_PORT (port.ml.r)

port	port value

Releases a message port that was assigned to the calling process.

BSL$DOWNLOAD_DEVICE ([handle.ma.r], [name.rt.dx1], file.rt.dx1)

handle	context identifier
name	programmable device name
file	VAX/VMS file name

Loads a VAX/VMS file containing device logic into a programmable device.

Procedure Parameter Notation

BSL$GET_DATA_INFO ([id.rl.r] [name.rt.dx1], item_list.ra.r)

id	data identifier code
name	data point name
item_list	pointer to array of item descriptors

Returns selected information about the data point to the calling program.

BSL$GET_DEVICE_ATTRIBUTES ([handle.ma.r], [name.rt.dx1], item_list.ra.r)

handle	context identifier
name	programmable device name
item_list	pointer to array of item descriptors

Returns selected generic attributes of the programmable device to the caller.

BSL$GET_SESSION_INFO (item_list.ra.r)

item_list	pointer to array of item descriptors

Returns information about the current terminal session to the caller.

Use as security check on TT, user, etc.

Procedure Parameter Notation

BSL$GET_SYSTEM_INFO ([id.rw.r], [name.rt.dx1], item_list.ra.r)

id	system internal identifier
name	application, device set or gateway name
item_list	pointer to array of item descriptors

Returns information about an application, device set, or gateway to the caller.

BSL$LOG_EVENT (code.rv.r, flag.rv.r, text.rt.dx1)

code	event type mask
flag	event flag mask
text	text string to log

Enters a user-defined text string in the System Audit file.

BSL$READ_DEVICE_DATA ([handle.ma.r], [name.rt.dx1], [address.rt.dx1], count.rw.r, buffer.ra.r)

GENERIC READ

handle	context identifier
name	programmable device name
address	starting memory location address
count	number of pieces of data to read, returns number actually read.
buffer	pointer to a buffer to receive the data read

Reads a buffer of data from the address in the device specified. Data will be returned in the same format as contained in the programmable device.

BSL$READ_DEVICE_STATUS ([handle.ma.r], [name.rt.dx1], count.ww.r, buffer.ra.r,)

handle	context identifier
name	programmable device name
count	number of bytes of data read
buffer	pointer to a buffer to receive data

Reads a buffer of programmable device specific status information.

Procedure Parameter Notation

```
BSL$RECEIVE_MESSAGE ( port.rl.r, [pointer.ra.r], [buffer.rx.r],
                      [src_port.wl.r], [code.ww.r], [size.ww.r],
                      [timeout.rq.r] )
```

port	port value
pointer	address of message
buffer	data buffer
src_port	port value
code	message code
size	size of data buffer in bytes
timeout	VAX/VMS binary delta time

Reads an interprocess message sent to the specified port. If no messages are pending, will wait until timeout value expires. Message may be referred to by pointer or data buffer.

```
BSL$SEND_MESSAGE ( [src_port.rl.r], [dest_port.rl.r], [pointer.ra.r],
                   [buffer.rx.r], [code.rw.r], [size.rw.r] )
```

src_port	port value
dest_port	port value
pointer	address of message
buffer	data buffer
code	message code
size	size of data buffer in bytes

Sends an interprocess message to the specified port. Message may be referred to by pointer or data buffer.

```
BSL$SET_DEVICE_ATTRIBUTES ( [handle.ma.r], [name.rt.dx1], item_list.ra.r )
```

handle	context identifier
name	programmable device name
item_list	pointer to array of item descriptors

Allows the calling process to set a generic attribute, such as the station number, of a programmable device.

Procedure Parameter Notation

BSL$SLEEP (string.rt.dx1)

string	VAX/VMS time formatted string

Causes the calling process to enter a LEF state until the specified time.

BSL$START_DEVICE ([handle.ma.r], [name.rt.dx1])

handle	context identifier
name	programmable device name

Starts the programmable device if the device was previously halted.

BSL$STOP_DEVICE ([handle.ma.r], [name.rt.dx1])

handle	context identifier
name	programmable device name

Stops the programmable device if the device was previously running.

BSL$UPLOAD_DEVICE ([handle.ma.r], [name.rt.dx1], file.rt.dx1)

handle	context identifier
name	programmable device name
file	VAX/VMS file name

Uploads logic from a programmable device into a VAX/VMS file.

BSL$WRITE_DEVICE_DATA ([handle.ma.r], [name.rt.dx1], [address.rt.dx1], count.rw.r, buffer.ra.r)

handle	context identifier
name	programmable device name
address	starting memory location address
count	number of pieces of data to write
buffer	pointer to a buffer containing data

Writes a buffer of data to the programmable device starting at the address specified. Data will be written in the same format as it appears in the caller's buffer.

Procedure Parameter Notation

BSL$WRITE_VERIFY_DEVICE_DATA ([handle.ma.r], [name.rt.dx1],
[address.rt.dx1], count.rw.r, buffer.ra.r)

handle	context identifier
name	programmable device name
address	starting memory location address
count	number of pieces of data to write
buffer	pointer to a buffer containing data

Writes a buffer of data to the programmable device starting at the address specified and verifies that it was written correctly. Data will be written in the same format as it appears in the caller's buffer.

Write, then read

APPENDIX B

SHOP FLOOR GATEWAY

SHOP FLOOR GATEWAY allows a PDP-11 processor to act as a front-end processor for communicating with shop floor programmable devices. A manufacturing application built on BASEWAY can use SHOP FLOOR GATEWAY to read and write to programmable devices on the shop floor.

The GATEWAY provides the actual communications interface to shop floor devices; BASEWAY provides application program communications and control functions. The GATEWAY offloads the automatic data-gathering and qualification processing from the VAX processor. All necessary protocol and data conversion are also performed by the GATEWAY.

GATEWAY software is downline loaded into a PDP-11 processor at startup using the VAX DECnet downline system load facility.

Information pertinent to adding new device support is given in Appendix D of this manual.

B.1 Features

The GATEWAY supports the BASEWAY programmable device access routines. These allow application programs to interact with shop floor devices in a variety of ways:

- Generic Access allows an application program to perform primitive functions through device-independent routines. These functions include reading and writing into device addresses, starting and stopping devices, and getting device attributes.

- Polled Access permits the GATEWAY to sample device data periodically and send a message to an application program when data changes.

B.2 Components

The Network Interface component allows the GATEWAY to participate in the BASEWAY messaging architecture.

The Event Processor performs data conversions to convert programmable device data into a format easily useable by an application program.

The Generic Server responds to generic device access and helps provide a generic interface to programmable controllers.

The Polled Server scans the programmable device point definitions and periodically collects data from programmable devices.

B.3 Functions

The SFG performs the following functions:

- o receives device-specific commands, acts upon them, and returns data and/or status.
- o polls programmable devices as specified in a polling database, performs certain preprocessing manipulation of the polled data, and transmits the polled data to BASEWAY.
- o detects PD communication faults and notifies BASEWAY of their occurrence.
- o performs self-diagnostic checks as well as diagnostic checks of programmable devices.
- o self-initializes following a bootstrap operation and runs without direct operator intervention.
- o accepts polling configuration data from BASEWAY on startup and keeps a copy of the "last polled" data in the polling database.

B.4 Device Access Supported

Several different methods of communicating with a programmable device are supported.

B.4.1 Direct Access

Direct access allows an application program anywhere in the network to perform logical "QIO" functions directly to the communications port. (NOTE: "QIO" functions are the RSX-11S executive directives for performing i/o to a device driver or device interface software. This permits application programs to perform device-specific functions that are not supported by other access methods.

B.4.2 Generic Access

Generic access allows applications programs to read registers, write status bits, and perform other control functions. No knowledge of the protocols is required.

5 Equipment Access

Equipment access allows the automatic gathering of predefined data. This data is collected in either a polled or unpolled manner. Once collected, the data is relayed to an application program.

B.6 Types of Data Capable of Being Polled

- Coils, or bits
- Registers, containing
  - bits
  - words
  - BCD
  - ASCII strings

APPENDIX C

DEVICE INTERFACE MODULES (DIMs)

C.1 Overview

Device Interface Modules (DIMs) contain the device-specific routines to handle the actual programmable device communications. The four main routines in a DIM and the tasks which call them are summarized in the table below:

ROUTINE	CALLED BY
i/o cancellation	DIRSRV, GENSRV, POLSRV
access / i/o initialization	DIRSRV, GENSRV, POLSRV
direct access request processing	DIRSRV
generic access request processing	GENSRV, POLSRV

The servers call the desired DIM routines by invoking the macro DIMDO$. This macro searches the Device Vector Table, D$VECT for the DIM entry table address associated with the interface device type. The server then uses the address contained in that DIM entry table at the offset of the routine. The DIM sets up and executes the necessary functions required to perform the desired request.

The PD access servers and the DIMs pass information through two data structures: the Line Access Block (LAB) and the Line Control Block (LCB). The LABs are used to maintain information related to the interface lines. The LCBs contain the context of service request to and from the DIM. These structures are described in more detail in a later section.

Device Interface Modules

Each PD server task has an active LAB for each defined PD interface. When service requests are received the LCBs are queued to the corresponding LAB. The LCBs are dequeued as the requests are processed. The following figures illustrate this interaction between the servers and the DIMs.

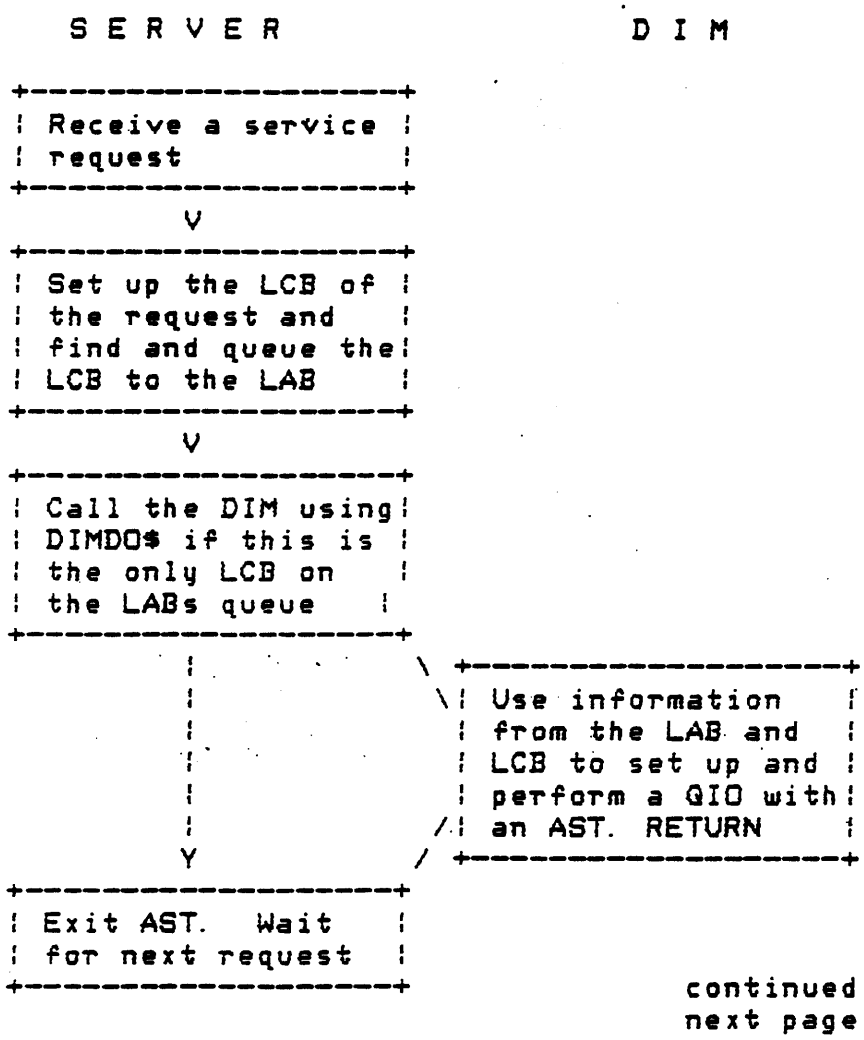


Figure 10A. PD Access Server - DIM Interaction

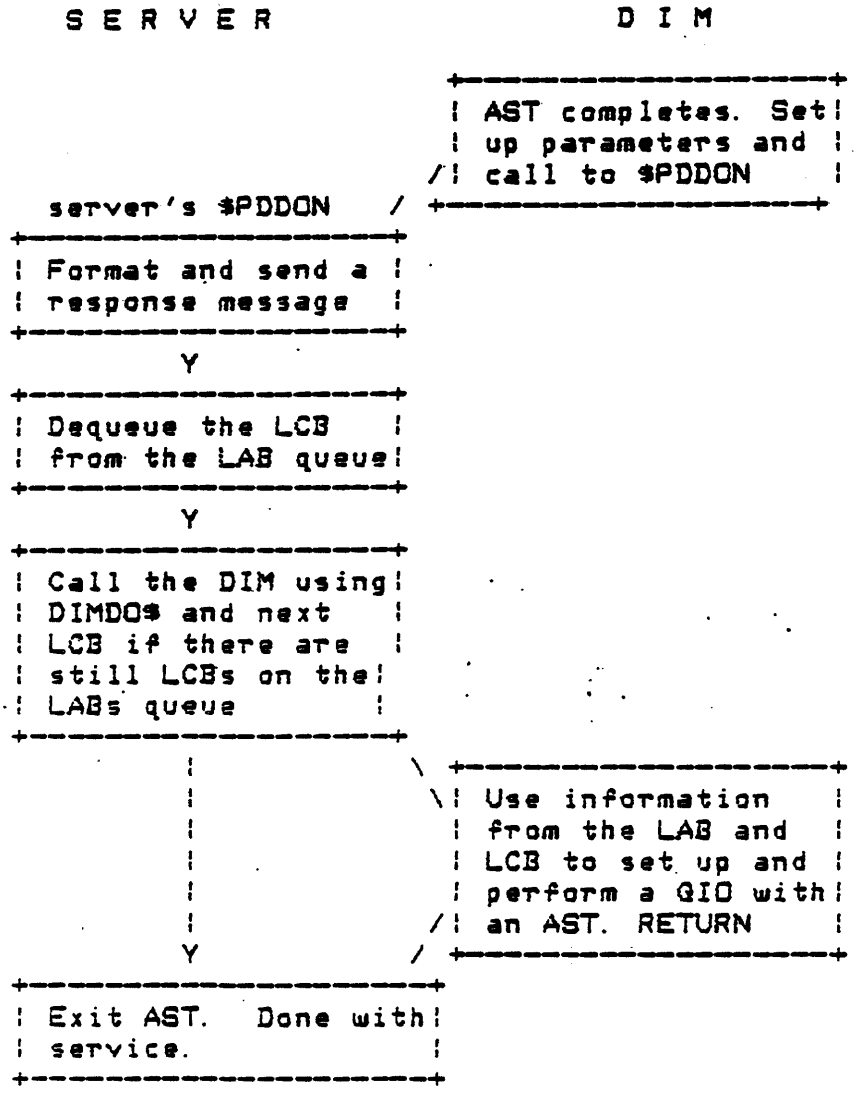


Figure 10B. PD Access Server - DIM Interaction (Continued)

C.2 DIM - PD Access Server Data Structures

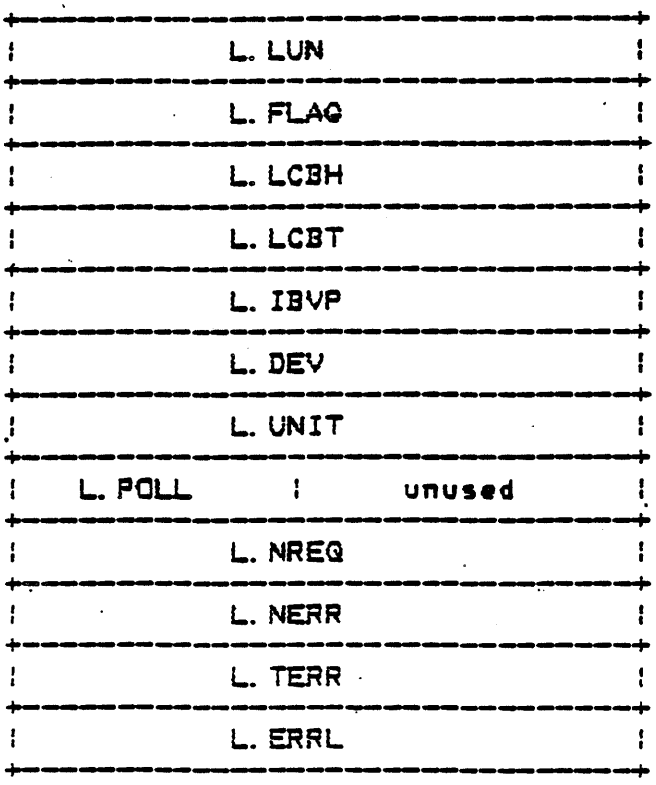
The PD Access Servers use two data structures to pass information to and from the DIMs. They are the Line Access Block (LAB) and the Line Control Block (LCB). These structures are passed to the DIMs in the following manner. The address of the LCB is passed to the DIM in R5; the address of the LAB associated with this LCB is contained in that LCB.

In the description of the data structure layouts that follow, the locations in the data structure are referred to by their symbolic offsets as defined in the corresponding macro as shown. Then, the usage of the location is given below using a special notation:

```
symbol (n) : ff  Description...

    symbol - symbolic offset from macro definition

         n - length of field in bytes

        ff - data format, "BU" - binary unsigned value
                          "BS" - binary (signed) value
                          "BM" - bit mapped (i.e., flags)
                          "C"  - coded
                          "A"  - ASCII
```

C.2.1 Line Access Blocks (LABs)

The Line Access Blocks contain PD interface line-related information required by the Access servers and the DIMs. Most of this information is copied from the Interface Definition Blocks (IBs) in VDR as the servers are started. The LAB offsets are defined in the macro LAB$ of MACROLIB and defined as follows:



L.LUN(2) : BU The logical unit assigned to this line by the server

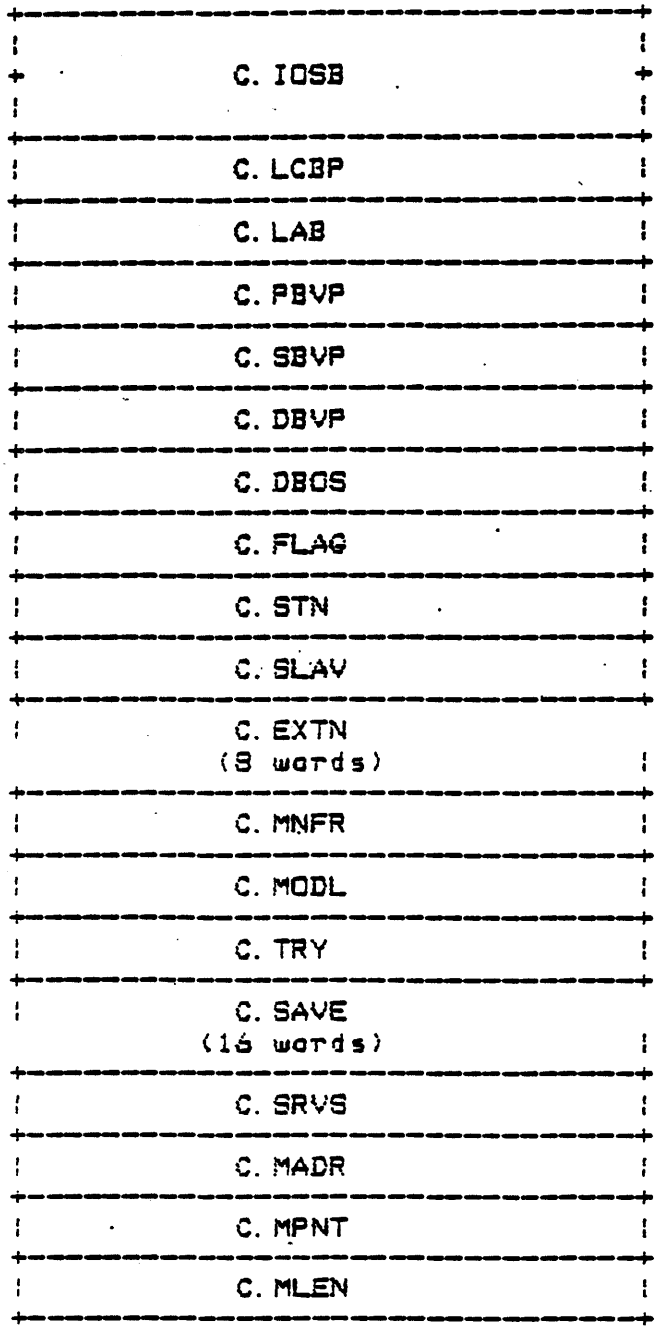
L.FLAG(2) : BM The status flag for the LAB's usage
- Bit 0 - LF.USD This block is used for an interface line
- Bit 1 - LF.ONL This block is associated with an accessible line/port
- Bit 2 - LF.PND An operation is pending on this line (i.e., there is an LCB queued to it)
- Bit 3 - LF.POL A polling cycle is in progress for this operation
- Bit 4 - LF.DED Dead poll flag

L.LCBH(2) : BU The queue head pointer of the pending LCB list

```
L.LCBT(2)       : BU The queue tail pointer of the pending
                     LCB list
 IBVP(2)        : BU The virtual pointer of the associated
                     IB in VDR
L.DEV(2)        : A The device name of this port
L.UNIT(2)       : BU The device unit number of this port
L.POLL(1)       : BU The poll counter, poll if 0
Unused(1)
L.NREQ(4)       : BS The count of requests serviced for
                     this port
L.NERR(4)       : BS The count of service error for this
                     port
L.TERR(4)       : BS The count of service timeout error
                     for this port
L.ERRL(4)       : BU The IOSB of the last failed QIO
```

C.2.2 Line Control Blocks (LCBs)

The context of the access service request is passed to the DIMs through the Line Control Blocks. The Line Access Blocks are defined in the macro LCB$ of the library MACROLIB and defined as follows:



A DIM has four functions:

- o cancellation / deaccess
- o initialization
- o direct access
- o generic access

All four routines need to return the following to the requesting server:

- completion service status code, offset C.SRVS of the LCB. The following status codes are defined:

  SS.SUC = service completed successfully
  SE.CAN = access service was cancelled
  SE.TMO = access service timed out
  SE.FNR = function rejected by service
  SE.OFL = interface was offline
  SE.PRO = protocol error
  SE.INT = interface error
  SE.DEV = invalid device type error
  SE.RSC = resource access error
  SE.VFY = verification error
  SE.CNT = invalid count (bit, byte, or word) for specific device QIO
  SE.ADR = address was invalid as determined by the DIM or device error return

- R5 remains pointing to the current Line Control Block
- R1 contains the number of bytes returned from the DIM completion.
- The status and byte count returned from the completed QIO needs to be placed in the C.IOSB and C.IOSB+2 offset of the LCB.

The four routines also receive and return operation-specific information (described in Sections B.4.1--B.4.4 below) from the requesting server. These fields are offsets of the LCB data structure and in the service specific message buffer. They are described in detail under the appropriate DIM routine.

The last three words of the Line Control Block contain the information required to access the service request message. All mailbox messages are contained in a common region GLBCOM. (See the

```
C.IOSB(4)       : BU i/o status block
C.LCBP(2)       : BU The pointer to the next LCB of a
                     LAB request queue
C.LAB(2)        : BU The pointer to the LAB that this
                     LCB is to service
C.FLAG(2)       : BM The block usage and status flag
C.PBVP(2)       : BU The programmable device block VP
C.SBVP(2)       : BU The polling set block VP
C.DBVP(2)       : BU The data definition block VP
C.STN(2)        : BU The device station address
C.SLAV(2)       : BU The device slave station address
C.EXTN(16)      : BU The extended address
C.MNFR(2)       : BU The manufacturer code
C.MODL(2)       : BU The model code
C.TRY(2)        : BU The number of attempts to try the
                     service before rejecting
C.SAVE(32)      : BU A user area to save i/o context for
                     ASTs
C.SRVS(2)       : BU The generic service status code of
                     the result of the service
C MADR(2)       : BU The mapped address of the message buffer.
                     This word is used for storing the
                     address of the message buffer.  This
                     buffer corresponds to the mapping
                     parameters in C.BPNT and C.BLEN.  This
                     address is only valid if the buffer is
                     currently mapped.
C.MPNT(2)       : BS The block offset into the global buffer
                     region of the message buffer.  This is
                     one of the parameters required  to map
                     the message buffer.
C.MLEN(2)       : BS The length of the message buffer in 32 word
                     blocks.  This is one of the parmeters
                     required to map the mailbox message buffer.
```

The actual format of the request messages depend on the server and the request.

C.3 DIM-Server Protocol and Considerations

The device access servers DIRSRV, GENSRV, and POLSRV pass information to and receive information from the DIMs through two data structures. These data structures are the Line Control Block (LCB) and the Line Access Block (LAB) described above. Before a server enters the device interface module, it places the address of the LCB in R5. The address of the LAB associated with the request is stored at location C.LAB(R5).

C.3.1 Initialization Procedure Considerations

This module performs any operations that may be necessary to initialize access to a device's interface software. An initialization request is performed by the servers before any device requests are acepted. These routines should be synchronous and use QIOWs where necessary.

The LCB associated with this request does not have a global message buffer associated with it. The initialization routine should pass the following to the routine $PDDON:

1. R5 = address of the LCB passed to the DIM with the following fields containing:
   - o C.SRVS = Generic status of the initialization routine
   - o C.IOSB = I/O status block of any QIO function that may have been performed.
2. R1 = 0 since there is no data to be returned.

C.3.2 Cancellation Procedure Considerations

The cancellation service routine allows the access servers to deaccess the communication software of the initialized interfaces. This is necessary if the servers need to update or reconfigure the device network. The servers perform a IO.KILL QIO function on each interface prior to calling the associated DIM cancellation service routine. The cancellation routine perorms all device specific operations required to deacess the interface.

As with the initialization routine, the LCB associated with the request does not have a message buffer. The cancellation routine should pass the following to the routine $PDDON:

1. R5 = address of the LCB passed to the DIM with the following fields containing:
   - o C.SRVS = Generic status of the initialization routine
   - o C.IOSB = I/O status block of any QIO function that may have been performed or filled with zeros.

RSX-11M/M-PLUS Executive Reference Manual for more information on virtual addressing and mapping) Inorder to address a message it must be first be mapped. The message buffer associated with the LCB can be mapped using the block offset pointer C.MPNT(R5) and the block length of the buffer C.MLEN(R5). It may be assumed that the servers have established the mapping context of the message buffer prior to calling the DIM.

The mapping context of the message buffer is not guaranteed when entering a completion AST. The SFG macro MPLCB$ may be used to map the message buffer and store the resulting address in the word C.MADR(R5). The macro ASDIM$ envokes MPLCB$ helping to reestablish the context of the request.

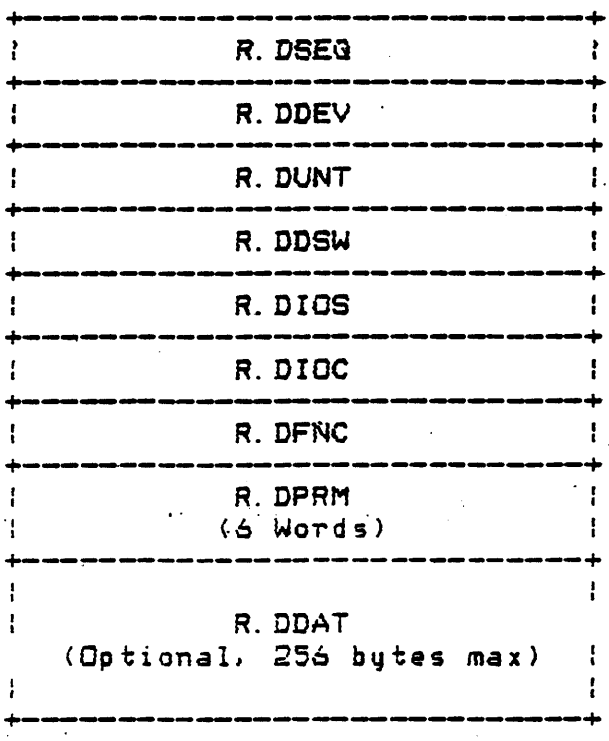
To prevent the loss of the message context and possible corruption of the Shop Floor Gateway system, DIM code should not modify the offsets C.MPNT and C.MLEN.

From time to time it may be necessary to allocate a buffer from memory. The servers are installed with extra free memory in their task partitions. Use the free memory head pointer F$REHD and the RSX System library routines $RQCB and $RLCB to allocate and deallocate the required buffers. Since the number of bytes returned by $RQCB may not match what was requested, be sure to store the address and actual length returned and use them to dealocate the buffer with $RLCB. These value can be stored in offsets to C.SAVE in the current LCB. Refer to IAS/RSX-11 System Library Routines Reference Manual for details on these routines.

C.3.3 Direct Access Service Processing Considerations

Direct access routines provide applications the ability to perform remote QIOs to programmable devices. The direct access message contains the information required by the DIM to set up the QIO directly. This message is described below.

QIOs performed by a direct access DIM routine should be asynchronous. That is, set up and issue the QIO specifying the next step in the logic as the AST entry point parameter and return. When the i/o completes continue processing the request as an AST routine.



R.DSEQ(2) : BU The sequence number of the request
R.DDEV(2) : BU The interface name
R.DUNT(2) : BU The interface unit number
R.DDSW(2) : BU The directive status word of the QIO
R.DIOS(2) : BU The i/o Status return of the QIO
R.DIOC(2) : BU The i/o Byte count
R.DFNC(2) : BU The i/o function code for the QIO
R.DPRM(12) : BU The 6-word device-specific QIO parameters
R.DDAT(256) : BU The PD data buffer

Direct Access Service Message Format

2. R1 = 0 since there is no data to be returned.

- Set up LCB, request message buffer and registers and call $PDDON to complete the request
- Exit the AST routine.

The direct access service message contains the following fields for the DIM's use:

- R.DFNC, the QIO function to be performed
- R.DPRM, an array of six words providing the necessary device-specific QIO parameter requirements for the DIM to issue a QIO to that device.

The following offsets in the direct access format Line Control Block need to be updated.

- C.SRVS, DIM completion service status code
- R.DDSW, the directive status word of the QIO invocation
- R.DIOS, the status return of the QIO
- R.DIOC, byte count returned from the QIO completion
- R.DDAT, buffer area holding data from a read QIO function

The following steps may be used as a guide to coding the direct service routines:

The logic flow of a typical direct service routine would be as follows:

- Validate function code and evaluate device dependent parameters in the R.DPRM offset of the LCB. Some parameters may need to be provided at the DIM level. These would include addresses of data buffers, DIM overwritten timeouts, or size parameters.
- Format and invoke the QIO specifying a completion AST routine.
- Test the directive status word of the QIO invocation, if successful, return to server, if unsuccessful, set the offsets necessary for the calling server, CALL $PDDON, and return to the server.
- Restore stack, set R5 as LCB pointer (AST parameter), and remap the message buffer.
- Evaluate the i/o status block of the resulting QIO function. and deterimine the corresponding generic status value. Retry if the function if was a timeout.

```
R.GBOF(1)       :  BU Bit off set field (device specific in its
                      usage and interpretation
  GCNT(2)       :  BU The data count
.. GDAT(256)    :  BU The PD data buffer
```

Generic Access Service Message Format

R.GSEQ is not currently beeing used and should not be modified by the DIM. R.GDID is used by the server to determine which LAB it should queue the request to, it too should not be modified by the DIM. R.GSTS is the generic completion code of the request. The sever's $PDDON copies the value that the DIM places in C.SRVS(R5) in R.GDID of the message.

The values in the fields M.CODE, R.GCNT and buffers R.GADR, R.GDAT are used by the DIM to perform the function. The value of M.CODE defines the function to be performed. Valid codes are defined in the macro MXCOD$ of MACROLIB. The are:

```
MG.RED Perform generic read from the device
MG.WRT Perform generic write to the device
MG.WTV Perform generic write to the device and verify
MG.RDS Perform read of device status
MG.SRD Start the device
MG.SPD Stop the device
MG.LON Log on to the device
MG.LOF Log off the device
MG.SUL Start the operation of uploading of device
       memory
MG.EUL End the operation of uploading of device memory
MG.SDL Start the operation of down loading of device
memory
MG.EDL End the operation of down loading of device
       memory
```

If the function requires an address in the device, it is passed in the buffer starting at offset R.GADR. The contents of this buffer is an internal BASEWAY address, that is, it was created by the BASEWAY device address translator. Both the translator and DIM code must interperate this buffer in the same way.

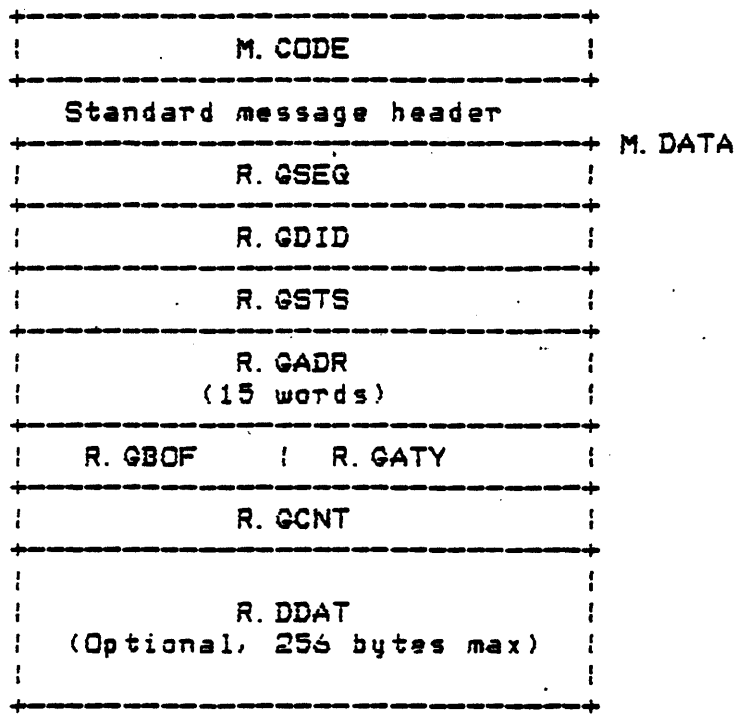
The R.GATY offset in the message determines the type of address contained in the buffer. Current address types are bit, physical, and logical. R.GBOF is used for device specific bit offset determination.

The R.GCNT field contains the number of data types to operate on. If R.GADR contains a bit address, R.GCNT is interms of bits. If R.GADR is a word register address, R.GCNT is the number of words to

C.3.4 Generic Access Service Processing Considerations

Generic access routine provide applications with a method of accessing devices generically. That is, an application may request to issue a read from a device and not need to provide the device specific information required to do the QIO. This implies that the generic access module of a DIM has the logic to translate a generic request code in to a series of steps required to set up, execute, and evaluate completion status in device-specific terms.

As with the other servers, the context of the request is passed to the DIM with R5 containing the address of a line control block. The LCB points to a generic request message. The information that is provided by a generic request message is shown in the figure below. These offsets are defined in the macro GENMS$ of MACROLIB.




```
M.CODE(2)       ;  BU The generic function code of the request
R.GSEQ(2)       :  BU The sequence number of the request
R.GDID(2)       :  BU The device logical identification
R.GSTS(2)       :  BU The generic service status return code
R.GADR(30)      :  BU The PD data address in internal BASEWAY
                      format
R.GATY(1)       :  C  The address type:
                        0  = Bit address
                        2  = Physical address
                        4  = Logical address
```

C.4 DIM - Utility Macros and Subroutines

..4.1 Code-Generating Macros

The following macros are defined in the DIM library for utility and coding clarity. They are quite helpful in writing Device Interface Modules. The following is a list of these macros.

ASDIM$ - Pop AST parameter (address of LCB into R5), save registers R0-R5, restore mapping context of message buffer associated with the LCB.
DTINI$ - Initialize device/function tables.
DTFNC$ - Create a function branch table entry.
DTMDL$ - Create a device model table entry.
DTMFR$ - Create a device manufacturer table entry.
ERTBL$ - Create a device-specific status to generic statu return translation table entry.
MPLCB$ - Map the message buffer associated with the current LCB.

C.4.1.1 ASDIM$ -

Form: ASDIM$

vokes the macros ASTSV$ and MPLCB$. ASTSV$ saves registers R0-R5 while it pops the AST parameter off the stack and puts it in R5. pops the AST parameter, the address of the LCB associated with the AST. MPLCB$ This macro pops an AST parameter off the stack and stores it in R5, saves the original contents of R0-R5 on the stack, remaps the cont

C.4.1.2 DTINI$ -

Form: DTINI$ <manufacturer table label>

Initializes the PSECTS for the manufacture, ..MFR, the model, ..MDL, and function branch, ..FNC tables.

use.

The actual processing of the generic request is accomplished by traversing through function branch tables. These tables are made up of entry points of routines, when called in the sequence of the table, will perform the function associated with that table. The function branch table table is determined by Manufacturer/model/function tables.

For each manufacturer supported by the DIM, there is a table of models associated with it. Each model is associated with a table which pairs the generic-function and addressing type (R.GATY in the generic message) with the beginning address of the proper function branch table. The macros DTINI$, DTFNC$, DTMDL$, and DTMFR$ described in section C.4.1 are used to build these tables. The subroutine $GENFC may be used to determine the function branch table for the current request.

The following steps should prove to be useful in designing and writing a generic access routine for a device interface module. The first step should be to analyze what steps are required to perform all the supported function for devices that are to use the DIM. Keeping in mind the following items.

- Because the service must support multiple interfaces at the same time, DIM code must be reentrant.
- Use the C.SAVE buffer in the LCB to maintain the context of the current QIO. These parameters may be needed if a retry for a timeout condition is desired.
- QIOs should be done with a completion AST routine.
- Translate device specific i/o status codes (success and error) into generic service status values. This is the only way that the servers will be able to determine succes or faliure.
- Deallocate any free memory buffers that may have been allocate during the processing of the function.
- Call $PDDON with the required parameters when terminating a function. $PDDON must be called whether the function was properly performed or not.

Refer to the sample DIM for a more detailed example of how to construct the generic routine.

C.4.1.5 DTMNF$ -
Form: DTMFR$ mfgr

mfgr = BASEWAY manufacturer code or 0

Creates an entry into the manufacturer table. The table entry consists of the manufacturer code in the first word and the current address of the PESECT ..MDL in the second word. This will become the first address of the next model table built using the DTMDL$ macro. A null parameter will generate a zero word and terminate the current manufacturer table. The manufacturer table is built in the PSECT ..MFR.

C.4.1.6 ERTBL$ -
Form: ERTBL$ deverr, generr

deverr = The device specific error code returned by the driver or ACP.

generr = The generic status code that the device error is to be translated to

Creates an entry into the generic error message table translation table. The entry is in the form of the device-specific i/o status code is in the first word and the generic status message is in the second word.

C.4.1.3 DTFNC$ - .

Form: DTFNC$ fnccod, [bitfnc], [phyfnc], [logfnc]

fnccod = generic function code

bitfnc = address of the bit operation function branch table. Zero if undefined.

phyfnc = address of physical operation function branch table. Zero if undefined.

logfnc = address of logical operations function branch table. Zero if undefined.

Creates an entry into the function branch table. The entry consists of the generic code in the first word. The address of an appropriate bit function branch table is in the second word. The third word is the appropriate physical operation entry point to its function branch table. The fourth word contains the entry point to the appropriate logical function branch. A null parameter for the macro will generate a zero word. This zero word will terminate the table. The function branch table is built in the PSECT ..FNC

C.4.1.4 DTMDL$ -
Form: DTMDL$ model

model = BASEWAY model code or 0

Creates an entry into the model table. This table entry consists of the model code in the first word and the current address of the PESECT ..FNC in the second word. This will become the first address of the next function branch table to be built using the DTFNC$ macro. A null parameter for the macro will generate a zero word and terminate the current model table. The model table is built in the PSECT ..MDL.

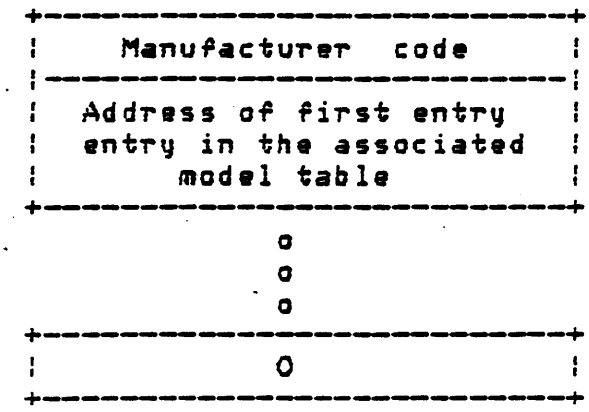
C.4.2.1 $GENFC -

Subroutine $GENFC - GENERIC FUNCTION TABLE LOOKUP MODULE

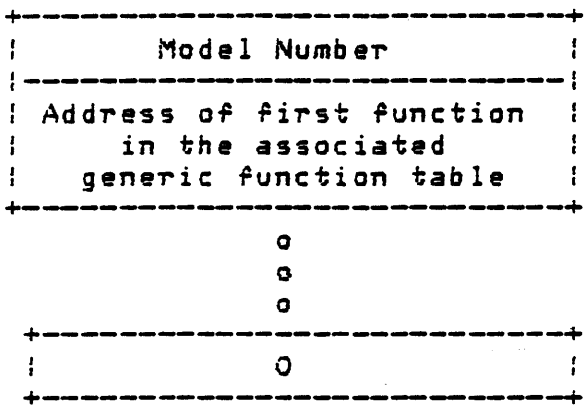
The purpose of this module is to perform table lookup operations determining the proper function branch table used to execute the generic function i/o request. The address of the function branch table is then passed back to the calling routine.

The associated tables are defined in the following manner:

Manufacturer table



Model table



C.4.2 DIM Subroutines

The following are utility subroutines that are used by DIMs.

$GENFC - Find function branch table
$PDDON - Server DIM completion routine

The following pages describe these routines in detail.

C.4.2.2 $PDDON -

The subroutine $PDDON is called by the DIM when it has completed one PD access service request, successfully or unsuccessfully. The primary functions of $PDDON are:

- o update service error statistics
- o return the service request response message to the requesting party
- o clean up and dispose of the current LCB
- o reenter the DIM if there are any more LCBs queued to the current LAB.

Because each server has different requirements and message formats, there is a $PDDON routine for each of them. In general the DIM needs to pass the following information to the $PDDON.

Calling Sequence:

```
CALL $PDDON
```

Input Parameters:

R1 = The number of data bytes that are to be returned to the requestor as a result of the service. Zero if there are none.

R5 = Address of the current LCB with the following fields modified

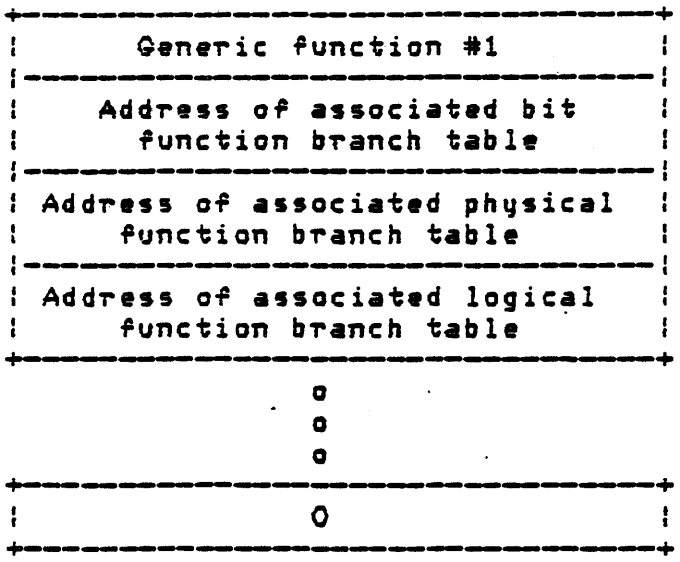
C.IOSB = i/o status block of the service i/o (should be zeroed by the DIM if no i/o was performed)

C.SRVS = Generic service status code of the service request

Output Parameters:

None Since the purpose of $PDDON is to clean up the current request context and establish the next, a DIM routine should clean up the stack and return or perform an AST exit.

Generic function table



The last entry in the manufacturer, model, and function table is a 0.

Library: DIMLIB.OLB

Calling Sequence:

```
CALL $GENFC
```

Input Parameters:

R0 Address of first entry in the Manufacturer function tables

R5 Address of current Line Control Block (LCB). It is assumed that the generic access message has been mapped and the address is stored in C.MADR(R5).

Output Parameters:

R0 Address of first function entry point in the resulting function branch table
R1 Address of first GIO function code to be issued.
R5 Preserved

C.5 Example DIM

```
;
;
;  Copyright (C) 1984
;  Digital Equipment Corp., Maynard, Mass.
;
;  This software is furnished under a license and may be used and  copied
;  only  in  accordance  with  the  terms  of  such  license and with the
;  inclusion of the above copyright notice.  This software or  any  other
;  copies  thereof may not be provided or otherwise made available to any
;  other person.  No title to and ownership of  the  software  is  hereby
;  transferred.
;
;  The information in this software is subject to change  without  notice
;  and  should  not  be  construed  as  a commitment by Digital Equipment
;  Corporation.
;
;  DIGITAL assumes no responsibility for the use or  reliability  of  its
;  software on equipment which is not supplied by DIGITAL.
;
;+
;  Subroutine $xxDIM - SAMPLE DEVICE INTERFACE MODULE
;
;       The xx Device Interface Module (DIM) is provided as a
;  template for users who wish to write their own DIM to support
;  programmable devices other than those already supported by
;  the SHOP FLOOR GATEWAY.
;
;       This template shows simple and concrete examples for writing
;  DIMs.  Explanations and code comments
;  guide the novice through the concepts and features of DIMs.
;  For the novice, the template builds a simple terminal server which
;  performs familiar TTy reads and writes, For the more advanced
;  coder, examples are given for actual PLC servers.  Additionally,
;  coding features and important points are highlighted for added
;  and improved functionality.
;
;       Generally, the template consists of descriptor tables
;  and four functional modules.
;
;       The descriptor tables define parameters like the manufacturer
;  of the device accessed by the DIM, the model number of the device,
;  the functions performed by the device, and the possible functions
;  for generic service.
;
;       The first module is the i/o cancellation module.  This module
;  is responsible for performing a disconnect operation from the
;  DIM-supported programmable device.  This operation may not be necessary
;  for a particular device type such as a terminal server.
```

```
;
;       The second module is the i/o initialization module.  This
;  module contains the code required to connect to the appropriate device
;  driver and/or ACP before accessing the communications interface(s).
;
;       The third module is the Direct Access Service module.  This
; module contains the code required to perform remote QIOs to the xx
; interface.
;
;       The fourth module is the Generic Access Service module. This
; module contains the code required to perform generic operations to
; xx devices.
;
;
; NOTE:  Before a Device access server can access a DIM, the
;        following must be done:
;
;        2) Assemble the module DIM source and insert into the
;           object library SFG$LIBRARY:DIMLIB.
;
;        3) Make an entry for this module in the device vector
;           table D$VECT (DEVVEC.MAC) by adding the following
;           line:
;
;               DEFDIM          xx      ; Add xx support
;
;        4) Reassemble and insert D$VECT into the object library
;           SFG$LIBRARY:DIMLIB.
;
;        5) Relink the Device access servers and move the
;           resulting images to SFG$SYSTEM with the following
;           VMS DCL commands:
;
;               $ SET DEFAULT SFG$ROOT:[SOURCE.POLSRV]
;               $ TKB @POLSRV.TKB
;               $ COPY POLSRV.TSK SFG$SYSTEM:
;               $ SET DEFAULT SFG$ROOT:[SOURCE.GENSRV]
;               $ TKB @GENSRV.TKB
;               $ COPY GENSRV.TSK SFG$SYSTEM:
;               $ SET DEFAULT SFG$ROOT:[SOURCE.DIRSRV]
;               $ TKB @DIRSRV.TKB
;               $ COPY DIRSRV.TSK SFG$SYSTEM:
;
;  Library: DIMLIB.OLB
;
;  Language: PDP-11 Macro
;
;-
;  Creation date:
;
;  Modification history:
```

```
;
;
;       .PAGE
;       .SBTTL  Macro declarations and definitions
;
;
;       .MCALL  LAB$,   LCB$,   GENMS$, DIRMS$, ASDIM$, SAVRG$
;       .MCALL  RESRG$, DTMDL$  GIO$S,  GATEP$, ASTX$S, SRVSC$
;       .MCALL  DEVCD$, MXCOD$, TBSIZ$, DTMFR$, DTFNC$, DTINI$
;       .MCALL  ERTBL$, GIOW$S
;
;
;       GATEP$          ; Gateway parameters
;       GENMS$          ; Define Generic Access message offsets
;       DEVCD$          ; Device names and model codes
;       SRVSC$          ; Generic service status codes
;       DIRMS$          ; Define Direct Access message offsets
;       MXCOD$          ; Define  mailbox message codes
;       LAB$            ; Define Line Access Block (LAB) offsets
;       LCB$            ; Define Line Control Block (LCB) offsets
```

```
        .PAGE
        .SBTTL  Data definitions

;=============================================================
;                  Device-specific i\o function table
;=============================================================
;
; Valid direct service QIO functions for the DIM-supported devices.
; xx.WRT, etc. are previously defined symbols.
; The following table is defined for validating QIO functions sent
; for direct servicing.
;
;-------------------------------------------------------------

xxVQIO:
        .WORD   IO.WLB          ; Logical write function
        .WORD   IO.WRT          ; Write function
        .WORD   IO.BWT          ; Bit write function
        .WORD   IO.RLB          ; Read function
        .WORD   IO.RED          ; Read function
        .WORD   IO.BRD          ; Bit read function
        .WORD   IO.DAG          ; Diagnostic read
        .WORD   IO.STP          ; stop device
        .WORD   IO.STR          ; start device
xxNBRQ = <.-xxVQIO>/2           ; Number of entries in table


;=============================================================
;                Generic error message code table
;=============================================================
;
; Creation of the error table associating IOSB error with generic
; error
;
;                 +----------------------+
;       Entry     |    i/o status error  |
;       number    |----------------------|
;        one      |  generic error code  |
;                 +----------------------+
;-------------------------------------------------------------

IOERR:
        ERTBL$  IS.SUC,SS.SUC   ; I/O successfully completed
        ERTBL$  IS.TMO,SE.TMO   ; Successful, but timeout
        ERTBL$  IE.TMO,SE.TMO   ; Timeout
        ERTBL$  IE.SRE,SE.PRO   ; Send reply message error
        ERTBL$  IE.OFL,SE.OFL   ; Off line
        ERTBL$  IE.DNR,SE.TMO   ; DZ timeout
        ERTBL$  IE.ABO,SE.CAN   ; Function canceled
        ERTBL$  IE.BCC,SE.TMO   ; Block check error
```

```
        TBSIZ$  IOERR,IOERCT    ; End of I/O error table
```

```
        .PAGE
;===============================================================
 .SBTTL Manufacturer/model function tables
;===============================================================
;       The manufacturer, model, and function tables are defined
;   here and used to process generic functions requested by the
;   user to the supported equipment.  Each manufacturer may have
;   several models; each model may support several generic
;   functions.  Those models which use identical functions tables
;   may share the same table.
;
;       Manufacturer/model names and codes are defined in the
;   macro DEVCD$.  For this example the manufacturer will have the
;   code MF.xxx and will have 3 models, MD.xxA, MD.xxB, and
;   MD.xxD.  Assume that MD.xxA and MD.xxB are similar devices and
;   use the same logic to perform their functions.
;
;---------------------------------------------------------------

        DTINI$  xxTBL           ; Initialize manufacturer table

        DTMFR$  MF.xxx          ; Start table for manf. XXX

        DTMDL$  MD.xxA          ; Make entries for modules MD.xxA
        DTMDL$  MD.xxB          ; and MD.xxB. Note that they
                                ; share the same function table

        DTFNC$  MG.RED,BITRD,PHYRD,LGCRD ; Read routines
        DTFNC$  MG.WRT,BITWR,PHYWR,LGCWR ; Write routines
        DTFNC$  MG.WTV,BITWV,PHYWV,LGCWV ; Write verify routines
        DTFNC$  MG.RDS,RDSTA    ; Status
        DTFNC$  MG.SRD,START    ; Start device
        DTFNC$  MG.SPD,STOP     ; Stop device
        DTFNC$  MG.SUL,UPREQ    ; Upline load request
        DTFNC$  MG.EUL,UPEND    ; End upline load
        DTFNC$  MG.SDL,DNREQ    ; Downline load request
        DTFNC$  MG.EDL,DNEND    ; End downline load

        DTMDL$  MD.xxC          ; Define the functions for the
                                ; model MD.XXC

        DTFNC$  MG.RED,CBITRD,PHYRD,CLGCRD ; Read routines
        DTFNC$  MG.WRT,CBITWR,PHYWR,CLGCWR ; Write routines
        DTFNC$  MG.WTV,CBITWV,PHYWV,CLGCWV ; Write verify routines
        DTFNC$  MG.RDS,RDSTA    ; Status
        DTFNC$  MG.SRD,START    ; Start device
        DTFNC$  MG.SPD,STOP     ; Stop device
        DTFNC$  MG.SUL,UPREQ    ; Upline load request
        DTFNC$  MG.EUL,UPEND    ; End upline load
        DTFNC$  MG.SDL,DNREQ    ; Downline load request
        DTFNC$  MG.EDL,DNEND    ; End downline load
```

```
        DTFNC$

        DTMDL$          ; Terminate the model table
        DTMFR$          ; Terminate manufacturer table
```

```
        .PAGE
;+
;============================================================
;       $ x x D I M  --  ENTRY POINT TABLE
;============================================================
;
;       This is the entry point table for the routines of this
; Device interface module.  The Device Access Service Tasks use
; this table to determine the entry points for appropriate DIM
; routine.  The order of the entry points is important since the
; Device Access servers reference them by their offset from $xxDIM.
; The order is:
;
;       .xxCAN Cancel routine transfer address
;       .xxINI Initialization routine transfer address
;       .xxDIR Direct access service routine transfer address
;       .xxGEN Generic access service routine transfer address
;
;------------------------------------------------------------

$xxDIM::                                ; MUST BE GLOBAL
        .WORD   .xxCAN                  ; Cancel service transfer address
        .WORD   .xxINI                  ; Initialization service transfer address
        .WORD   .xxDIR                  ; Direct access service transfer address
        .WORD   .xxGEN                  ; Generic access service transfer address
```

```
        .PAGE
        .SBTTL  .xxCAN  - cancellation service routine
;+
;=============================================================
;       .  x x C A N  --  SERVICE CANCELLATION SERVICE ROUTINE
;=============================================================
;
;       This module performs the necessary actions required to
; deaccess the device's networking interface.
;
; The following QIO function code is device specific.  Depending
; on the type of device, a different type of QIO function may
; need to be used, or the deaccess QIO function or QIO
; invocation may not be needed at all for your particular device.
;
; Input:
;       R5 =    Address of Line Control Block (LCB)
;               containing the context of the cancel
;               request.  Offsets are:
;
;          C.LAB    = Address of the Line Access Block (LAB)
;                     of the interface to perform the cancel
;                     request on
;
; Input to $PDDON:
;       R1 = 0 since there is not data to be returned to any
;            one
;       R5 = Address of the current LCB with offsets:
;
;          C.IOSB   =  I/O status block of the i/o deaccess
;                      operation(s)
;          C.SRVS   =  Service status code sent back to servers
;                      SS.SUC for successful completion other
;                      codes the server may expect are located
;                      in the macro SRVSC$.
;
; EFFECT:
;
;-------------------------------------------------------------
```

```
        .ENABL  LSB
xxCAN:

        SAVRG$  <R4>
        MOV     C.LAB(R5),R4        ; R4 points to LAB this block is queued

        QIOW$S  #IO.DAC,L.LUN(R4),#EF.PD,,R5 ; Deaccess LUN
        BCC     20$                 ; BR if directive successful

        MOV     $DSW, C.SRVS(R5)    ; use DSW$ error for generic
                                    ; code
        BR      40$                 ; and return.

; Determine what what generic status value to return from the
; IOSB and the IO error table.

20$:    MOV     #IOERR,R1           ; R1 points to i/o status error table
        MOV     #IOERCT,R2          ; R2 points to Size of error table
25$:    CMPB    C.IOSB(R5),(R1)     ; Determine i/o status
        BEQ     30$                 ; go to insert generic function
        ADD     #2,R1               ; Try next entry
        SOB     R2,25$              ; Do until end of table
        MOV     #SE.FNR,C.SRVS(R5)  ; Function error
        BR      40$
30$:
        MOVB    1(R1),R1            ; Move in error
        MOV     R1,C.SRVS(R5)       ; Insert generic status

 $:     CLR     R1                  ; Error - zero byte cnt
        CALL    $PDDON              ; clean up operations
        RETURN

        .DSABL LSB
```

```
        .PAGE
        .SBTTL  .xxINI -  initialization service routine
;+
;=============================================================
;          . x x I N I  --  SERVICE INITIALIZATION ROUTINE
;=============================================================
;
;
;       This module performs the necessary actions required to
; initialize access to an xx interface line
;
; Input:
;       R5 = Address of Line Control Block (LCB) containing
;               initializaton request parameters:
;          C.LAB    =  Address Line Access Block (LAB) to perform
;                        initialization
;
;  Output to PDDON$:
;
;       R1 = 0 since there is no data to be returned
;
;       R5 = Address of LCB used in this operation passing back
;               the offsets containing following information:
;
;          C.IOSB   =  I/O status block of the i/o deaccess
;                        operation(s)
;          C.SRVS   =  Service status code sent back to servers
;                        SS.SUC for successful completion other
;                        codes the server may expect are located
;                        in the macro SRVSC$.
;-------------------------------------------------------------
```

```
        .ENABL LSB

.xxINI:
        SAVRG$  <R1,R2,R4>      ; save and restore registers
        MOV     C.LAB(R5),R4    ; get LAB for this block

; Issue access QIO

        QIOW$S  #IO.ACW,L.LUN(R4),#EF.PD,,R5
        BCC     20$             ; BR if directive successful
        MOV     $DSW, C.SRVS(R5) ; use DSW$ error for generic
                                ; code
        BR      40$             ; and return.

; Determine what what generic status value to return from the
; IOSB and the IO error table.

20$:
        MOV     #IOERR,R1       ; R1 points to i/o status error table
        MOV     #IOERCT,R2      ; R2 points to Size of error table
25$:
        CMPB    C.IOSB(R5),(R1) ; Determine i/o status
        BEQ     30$             ; go to insert generic function
        ADD     #2,R1           ; Try next entry
        SOB     R2,25$          ; Do until end of table
        MOV     #SE.FNR,C.SRVS(R5) ; Function error
        BR      40$
30$:
        MOVB    1(R1),R1        ; Move in error
        MOV     R1,C.SRVS(R5)   ; Insert generic status

40$:    CLR     R1              ; Error - zero byte count
        CALL    $PDDON          ; clean up operations
        RESRG$  <R4,R2,R1>      ; save and restore registers
        RETURN
        .DSABL LSB
```

```
        .PAGE
        .SBTTL .xxDIR   -  direct access service routine
;+
;=============================================================
;    . X X D I R  --  DIRECT ACCESS SERVICE PROCESSING ROUTINES
;=============================================================
;
;       This module provides the actions required to perform a
; direct access QIO as defined in the direct access service
; request message.
;
;  This direct access service routine performs the following:
;
;   1) Validates the QIO function code sent with the QIO
;        function code table (xxVQIO)
;
;   2) Validates the device dependent parameters sent in the
;        service request message starting at R.DPRM
;
;   3) Sets up the directive QIO function with the QIO
;        completion AST parameter set to #xxDAST
;
;   4) Determines if QIO was invoked successfully.  If the QIO
;        was unsuccessfully dispatched, sets proper status codes
;        and calls $PDDON to clean up before returning to server.
;
;   5) Saves registers and restores mapping context of the
;          mailbox message with the ASDIM$ macro
;
;   6) Depending on the type of device supported, checks for time
;          outs or block check errors, then retries the QIO the number
;          of times indicated in the C.TRY offset of the LCB.
;
;   7) Determines success or failure of the i/o completion, sets up
;        the following offsets in the LCB and request message.
;
;              - C.SRVS - generic status message code determined
;                         from i/o error to the generic status routine
;              - R.DIOS - i/o status from C.IOSB
;              - R.DIOC - i/o byte count from C.IOSB+2
;
;   8) Determines if a read was invoked, if so, sends the byte
;          count in R1 to PDDON.  R5 still must point to the
;          beginning of the current LCB.
;
;   9) Calls $PDDON
;
;   10) Restores registers in the following order with the RESRG$
;          macro:  <R0,R1,R2,R3,R4,R5>
;
```

```
; Input:
;
;       R5 =    Address of Line Control Block (LCB)
;               containing the context of the cancel
;               request. Offsets are:
;
;          C.LAB  = Address of the Line Access Block (LAB)
;                   of the interface to perform the cancel
;                   request on. The primary field that is
;                   used from this block is the logical unit
;                   number at L.LUN.
;
;          C.MPNT = Block pointer to the direct service
;                   request message.  This value is used in
;                   mapping the message in the servers task
;                   space.
;
;          C.MLEN = Length of direct service request message
;                   in number of blocks.  This is also used in
;                   the mapping of the message.
;
;          C.MADR = Mapped address of the direct service
;                   request message. This address is only
;                   valid if the message has been mapped. It
;                   is mapped when the server calls this
;                   module at the entry point xxDIR. It must
;                   be remapped when the executing completion
;                   AST or if anything else destroys the
;                   mapping context.  The format and values
;                   of the message follow (offsets defined
;                   in macro DIRMS$).
;
;       R.DSEQ  =  Request sequence number.
;
;       R.DDEV  =  xx (Interface name in ASCII).
;
;       R.DUNT  =  Interface unit number.
;
;       R.DFNC  =  QIO function code to use for the request.
;
;       R.DPRM  =  Array of 6 words containing the
;                  device-specific optional parameters for
;                  the QIO.
;
;       R.DDAT  =  Beginning of the request data buffer.
```

```
;
; Output to PDDON$:
;
;       R1 =    Number of data bytes to return to requestor
;               starting at offset R.DDAT in the direct
;               service message.
;
;       R5 =    Address of LCB used in this operation passing
;               back the offsets containing following
;               information:
;
;          C.IOSB = I/O status block of the i/o deaccess
;                   operation(s).
;          C.SRVS = Service status code sent back to servers
;                   SS.SUC for successful completion) other
;                   codes the server may expect are located
;                   in the macro SRVSC$.
;
;          C.MLEN = Block length of the message buffer.
;          C.MPNT = Block pointer of the message buffer.
;          C.MADR = Mapped address of the direct service
;                   with the following offsets updated:
;
;                     R.DDSW  =     Directive status word of QIO
;                                   invocation.
;                     R.DIOC  =     I/O count in bytes that resulted
;                                   from the QIO.
;                     R.DDAT  =     Beginning of data data to be
;                                   returned buffer.
;
;-
```

```
        .ENABL LSB

xDIR::

        SAVRG$  <R4,R3,R2>          ; Save registers
        MOV     C.MADR(R5),R3       ; R3 points to Beginning of direct data
        MOV     R.DFNC(R3),C.SAVE(R5) ; Direct QIO function code
        MOV     R3,R.DPRM(R3)       ; Beginning of data message
        ADD     #R.DDAT,R.DPRM(R3) ; Address of data buffer
        MOV     R3,C.SAVE+2(R5) ; Beginning of data message
        ADD     #R.DPRM,C.SAVE+2(R5) ; Address of parameter block

; Determine if QIO function code is valid

        MOV     #xxNBRQ,R2          ; R2 points to Number of xx QIOs
        MOV     #xxVQIO,R4          ; R4 points to Beginning of xx QIO functions
5$:     CMP     R.DFNC(R3),(R4)+ ; Find match in table
        BEQ     10$
        SOB     R2,5$               ; No match - try next entry

; Bad QIO function from direct access server mailbox

        MOV     #SE.FNR,R.DDSW(R3) ; Simulate bad directive
        CLR     C.IOSB(R5)          ; Clear 1st word IOSB
        CLR     C.IOSB+2(R5)        ; Clear IOSB byte count
        BR      DIRERR

; Test size of message buffer

10$:    TST     R.DPRM+2(R3)        ; Test for neg or zero count
        BLE     20$                 ;
        CMP     R.DPRM+2(R3),#R.DLDT ; Test greater than max size
        BLE     DIRQIO              ; Perform QIO

; Invalid length of buffer

20$:
        MOV     #IE.IBS,R.DDSW(R3) ; Move inv buff count message
        MOV     #SE.CNT, C.SRVS(R5) ; Bad count status code
        CLR     C.IOSB(R5)          ; Clear 1st word of IOSB
        CLR     C.IOSB+2(R5)        ; Clear byte count in IOSB
        BR      DIRERR

; Issue QIO, R2 holds address to the six device-specific
; parameters

DIRQIO:

        MOV     #SE.FNR,C.SRVS(R5) ; Assume error code
        MOV     C.SAVE(R5),R2       ; R2 points to Device param block
        MOV     C.LAB(R5),R4        ; Get LAB for this block
```

```
        MOV     L.LUN(R4),R4    ; Need space on QIO line

 QIO$S  C.SAVE(R5),R4,,,R5,#xxDAST,<(R2),2(R2),4(R2),6(R2),8.(R2),10.(R2)>

        MOV     $DSW,R.DDSW(R3) ; Save directive status word
        BPL     DIREND

; Set up status codes before returning with QIO failure
; R3 points to start of data area

DIRERR:
        MOV     C.IOSB(R5),R.DIOS(R3) ; Save i/o status
        CLR     R.DIOC(R3)      ; Zero i/o byte count
        CLR     R1              ; R1 points to number of bytes read
        CALL    $PDDON          ; Clean up

; End of Direct access service processing

DIREND:
        RESRG$  <R2,R3,R4>      ; Restore registers
        RETURN
        .DSABL LSB
```

```
        .ENABL LSB

xxDAST:
;-------------------------------------------------------------
;
;       This is the i/o completion AST routine for a Direct service
; request.  It is responsible for evaluating the completion of the
; i/o operation, setting up the parameters required by $PDDON,
; and calling $PDDON to return the request and clean up for the
; next request.
;
; Inputs:
;
;       (SP) = AST parameter, the IOSB of the QIO function.
;              This address also points to the offset C.IOSB
;              of the LCB associated with the QIO.
;
;       AST parameter is popped from stack and all registers
;       are preserved .
;
;-------------------------------------------------------------

        ASDIM$                      ; Pop AST parameter (address
                                    ; of request's LAB) and put
                                    ; into R5 and  save registers
                                    ; R0-R5 on the stack.
                                    ; Remap the message buffer
                                    ; referenced by the LCB (map
                                    ; C.MLEN(R5) blocks starting
                                    ; at block C.MPNT(R5)) and
                                    ; store the resulting address
                                    ; in C.MADR(R5).

; Determine if there has been a time out or block check

        CMPB    C.IOSB(R5),#IE.TMO ; Test for time out condition
        BEQ     10$                ; Yes, decrement count and retry
        CMPB    C.IOSB(R5),#IE.BCC ; Test block check condition
        BNE     20$                ; No, determine status and condition
10$:    DEC     C.TRY(R5)          ; Number of retries
        BLE     20$                ; Done trying - continue
        CALL    DIRQIO             ; Retry direct service QIO
        BR      100$               ; Go to end of AST

; Determine generic message/error response

20$:    MOV     C.MADR(R5),R3      ; R3 points to Beginning of data
        MOV     #IOERR,R1          ; R1 points to Error table entry point
        MOV     #IOERCT,R2         ; R2 points to Number of entries in
                                   ; error table
```

```
30$:    CMPB    C.IOSB(R5),(R1) ; Determine i/o status
        BNE     40$             ; Didn't match, go set up next

; Found i/o status return code, send equivalent generic error message

        MOVB    1(R1),R1        ; Byte generic error
        MOV     R1,C.SRVS(R5)   ; Insert generic status
        BR      50$             ; Go to determine byte count

; Point to next entry in error table and try again
; If at end of table use default error message and continue

40$:    ADD     #2,R1           ; Try next entry
        SOB     R2,30$          ; End of table?
        MOV     #SE.FNR,C.SRVS(R5) ; Default generic error message

; Set up number of bytes to return, send to $PDDON

50$:    CLR     R1              ; Zero for other than read
        MOV     C.MADR(R5),R3   ; R3 points to Beginning of data
        CMP     R.DFNC(R3),#xx.RUP      ; Test for read function
        BEQ     55$             ; Read, move number of bytes read
        CMP     R.DFNC(R3),#xx.RPR      ; Test for read function
        BEQ     55$             ; Read, move number of bytes read
        CMP     R.DFNC(R3),#xx.DAG      ; Diagnostic read
        BNE     60$
55$:    MOV     C.SAVE+2(R5),R1 ; R1 points to Number of data bytes

; Save status and byte count then return

60$:
        MOV     C.IOSB(R5),R.DIOS(R3)   ; Save i/o status
        MOV     C.IOSB+2(R5),R.DIOC(R3) ; Save i/o byte count
        CALL    $PDDON          ; Perform direct access cleanup

; Exit direct service AST

100$:   RESRG$  <R0,R1,R2,R3,R4,R5>     ; Restore registers
        ASTX$S                          ; Exit AST (param already popped)

        RETURN
        .DSABL LSB
```

```
        .PAGE
        .SBTTL .xxGEN   -  generic access service routine

;*****************************************************************
;       .  x  x  G  E  N  --  GENERIC ACCESS SERVICE PROCESSING ROUTINES
;*****************************************************************
;
;       This module accepts a generic access request, determines
;  and executes the device specific steps required to perform the
;  request.
;
;       The generic functions are defined and specified at the
;  beginning of the DIM in the  manufacturer/model tables (xxTAB).
;  These functions are composed of a sequence of subfunctions
;  defined below.
;
;
;     The generic processing routine executes in the following order:
;
;     1) Set up entry point to manufacturer/model table in R0
;
;     2) Call $GENFC - finds associated function branch table for
;        the generic service requested.
;
;     3) Determine if $GENFC returned with an error (R0 is negative)
;         - error will be due to a model number or manufacturer
;           requested in the QIO was not found in the manufacturer/
;           model table.
;
;     4) If no error occurred, begin processing the generic request
;        by executing the code pointed to by each entry point
;        listed in the appropriate function branch table
;        (determined by  $GENFC).  R0 contains the address of the
;        first function entry point.  R1 contains the address of
;        the first QIO function code.
;
;
```

```
;
;  INPUT:
;
;        R5  =   Address of the Line Control Block (LCB)
;                containing the context of a generic access
;                service request. Applicable offsets are:
;
;            C.LAB  =    Address Line Access Block (LAB) of the
;                        interface that the device is located on.
;            C.STN  =    Station address of the device
;            C.SLAV =    Slave station address of device
;            C.EXTN =    Extended address
;            C.MNFR =    Manufacture code of the device
;            C.MODL =    Model code of the device
;            C.MLEN = Block length of the message buffer.
;            C.MPNT = Block pointer of the message buffer.
;            C.MADR =    Address of the beginning of the mailbox
;                        message containing the request.  Also
;                        the address for the response.. The incoming
;                        message is formatted as follows:
;
;                R.GADR =     Data Address of the programmable
;                             device
;                R.GCNT =     Count of data to use in i/o
;                R.GDAT =     Optional device data buffer of up
;                             to 256 bytes. Use of buffer is
;                             dependent on the request (read or
;                             write)
;
;        Information passed through the associated LAB is:
;
;        L.LUN   =  Logical Unit Number (LUN) to use in performing
;                   i/o on the device interface.
```

```
;
; Output to PDDON$:
;
;       R1 =    Number of data bytes to return to requestor
;               starting at offset R.GDAT in the direct
;               service message.
;
;       R5 =    Address of LCB used in this operation passing
;               back the offsets containing following
;               information:
;
;          C.IOSB = I/O status block of the i/o deaccess
;                   operation(s).
;          C.SRVS = Service status code sent back to servers
;                   SS.SUC for successful completion) other
;                   codes the server may expect are located
;                   in the macro SRVSC$.
;          C.MLEN = Block length of the message buffer.
;          C.MPNT = Block pointer of the message buffer.
;          C.MADR = Mapped address of the direct service
;                   with the following offsets updated:
;
;                R.GDAT  =     Beginning of data data to be
;                              returned buffer.
```

```
;
;    C.SAVE will be used in the following way.  The programmer
;  may use them in any way that makes life easy.
;
;    C.SAVE      = Address of current QIO function
;    C.SAVE+2    = Address of data buffer
;    C.SAVE+4    = Count (bits,bytes or words) requested
;    C.SAVE+6    = Address of PD address buffer (allocated)
;    C.SAVE+8.   = Subfunction code
;    C.SAVE+10.  = Current completion AST address
;    C.SAVE+12.  = Assigned when new buffer allocated
;                  (previous buffer addr)
;    C.SAVE+14.  = Temporary storage for computed byte count
;                  (bit functions)
;    C.SAVE+16.  = Actual length of address buffer
;    C.SAVE+18.  = Actual length of buffer 1          [2]
;    C.SAVE+20.  = Actual length of buffer 2          [12]
;    C.SAVE+30.  = Error exit branch
;
;-

;-------------------------------------------------------------
;                  Function branch tables
;-------------------------------------------------------------


; Read register function entry points
LGCRD:                                  ; Logical Read of Register
        .WORD   1                       ; Issue one QIO function
        .WORD   IO.RLB                  ; Logical read
        .WORD   LCBINI                  ; Initial register setup
        .WORD   CNVBYT                  ; Convert registers to bytes
        .WORD   BYTCHK                  ; Validate byte count
        .WORD   GENQIO                  ; Issue requested QIO function
        .WORD   GENEND                  ; End generic function the next
                                        ; word contains the next jump
                                        ; after entering the AST
        .WORD   QIOEXM                  ; Test for successful QIO
        .WORD   CLNUP                   ; Success clean up, call PDDON
        .WORD   ASTEXT                  ; Exit AST

CLGCRD:                                 ; Logical Read of Register for
                                        ; XXC devices
        .WORD   1                       ; Issue one QIO function
        .WORD   IO.RLB                  ; logical read
        .WORD   LCBINI                  ; Initial register setup
        .WORD   ALABUF                  ; Allocate address buffer
        .WORD   CNVBYT                  ; Convert registers to bytes
        .WORD   BYTCHK                  ; Validate byte count
        .WORD   GENQIO                  ; Issue requested QIO function
```

```
        .WORD   GENEND          ; End generic function the next
                                ; word contains the next jump
                                ; after entering the AST
        .WORD   QIOEXM          ; Test for successful QIO
        .WORD   CLNUP           ; Success clean up, call PDDON
        .WORD   ASTEXT          ; Exit AST


; Write register generic function entry points

LGCWR:
        .WORD   1               ; One QIO operation
        .WORD   IO.WLB          ; Logical write
        .WORD   LCBINI          ; Initial register setup
        .WORD   CNVBYT          ; Convert register cnt to bytes
        .WORD   BYTCHK          ; Validate byte count
        .WORD   GENQIO          ; Issue requested QIO function
        .WORD   GENEND          ; End generic function
                                ; word contains the next jump
                                ; after entering the AST
        .WORD   QIOEXM          ; Test for successful QIO
        .WORD   CLNUP           ; Success clean up, call PDDON
        .WORD   ASTEXT          ; Exit AST


GCWR:   .WORD   1               ; One QIO operation
        .WORD   IO.WLB          ; Logical write
        .WORD   LCBINI          ; Initial register setup
        .WORD   ALABUF          ; Allocate address buffer
        .WORD   CNVBYT          ; Convert register cnt to bytes
        .WORD   BYTCHK          ; Validate byte count
        .WORD   GENQIO          ; Issue requested QIO function
        .WORD   GENEND          ; End generic function
                                ; word contains the next jump
                                ; after entering the AST
        .WORD   QIOEXM          ; Test for successful QIO
        .WORD   CLNUP           ; Success clean up, call PDDON
        .WORD   ASTEXT          ; Exit AST


; Write/verify register generic function entry points

LGCWV:                          ; Write/verify register gen fnc
        .WORD   2               ; Two functions performed
        .WORD   IO.WLB          ; Logical write
        .WORD   IO.RLB          ; Logical read
        .WORD   LCBINI          ; Initial register setup
```

```
        .WORD   CNVBYT          ; Convert count to bytes
        .WORD   BYTCHK          ; Validate byte count
        .WORD   GENQIO          ; Issue write function
        .WORD   GENEND          ; End write processing next
                                ; word contains address of the
                                ; next jump returning from AST
        .WORD   QIOEXM          ; Test for successful QIO
        .WORD   ALVBUF          ; Allocate verify buffer
        .WORD   GENQIO          ; Issue the Read QIO
        .WORD   ASTEXT          ; Exit this level of AST next
                                ; word contains address of next
                                ; jump returning from address
        .WORD   QIOEXM          ; Test for successful QIO
        .WORD   CMPBUF          ; Compare two buffers
        .WORD   DALBUF          ; Deallocate dynamic buffer
        .WORD   CLNUP           ; Success! Clean up and call PDDON
        .WORD   ASTEXT          ; Exit AST done with function

CLGCWV:                         ; Write/verify register gen fnc
        .WORD   2               ; Two functions performed
        .WORD   IO.WLB          ; Logical write
        .WORD   IO.RLB          ; Logical read.
        .WORD   LCBINI          ; Initial register setup
        .WORD   ALABUF          ; Allocate address buffer
        .WORD   CNVBYT          ; Convert count to bytes
        .WORD   BYTCHK          ; Validate byte count
        .WORD   GENQIO          ; Issue write function
        .WORD   GENEND          ; End write processing next
                                ; word contains address of the
                                ; next jump returning from AST
        .WORD   QIOEXM          ; Test for successful QIO
        .WORD   ALVBUF          ; Allocate verify buffer
        .WORD   GENQIO          ; Issue the Read QIO
        .WORD   ASTEXT          ; Exit this level of AST next
                                ; word contains address of next
                                ; jump returning from address
        .WORD   QIOEXM          ; Test for successful QIO
        .WORD   CMPBUF          ; Compare two buffers
        .WORD   DALBUF          ; Deallocate dynamic buffer
        .WORD   CLNUP           ; Success! Clean up and call PDDON
        .WORD   ASTEXT          ; Exit AST done with function

PHYRD:                          ; Physical Read Register
        .WORD   1               ; Issue one QIO function
        .WORD   IA.RED          ; Physical read
        .WORD   LCBINI          ; Initial register setup
        .WORD   CVTBYT          ; Convert register count to bytes
        .WORD   VLDBYT          ; Validate byte count
        .WORD   GENQIO          ; Issue QIO requested function
```

```
        .WORD   GENEND          ; End generic function, next
                                ; word contains the next jump
                                ; after entering the AST
        .WORD   QIOEXM          ; Test for successful QIO
        .WORD   CLNUP           ; Success clean up, call PDDON
        .WORD   ASTEXT          ; Exit AST

; Physical writes for download operations

PHYWR:                          ; Physical Write Register
        .WORD   1               ; One QIO for physical writes
        .WORD   IA.WRT          ; Physical write
        .WORD   LCBINI          ; Initial register setup
        .WORD   CNVBYT          ; Convert register cnt to bytes
        .WORD   BYTCHK          ; Check byte count
        .WORD   GENQIO          ; Issue requested QIO
        .WORD   GENEND          ; End generic function
                                ; word contains the next jump
                                ; after entering the AST
        .WORD   QIOEXM          ; Test for successful QIO
        .WORD   CLNUP           ; Success clean up, call PDDON
        .WORD   ASTEXT          ; Exit AST

; Write/verify register generic function entry points

PHYWV:                          ; Write/verify register gen fnc
        .WORD   2               ; Two functions performed
        .WORD   IA.WRT          ; Privileged write function
        .WORD   IA.RED          ; Privileged read function
        .WORD   LCBBINI         ; Initial register setup
        .WORD   CNVBYT          ; Convert register count to bytes
        .WORD   BYTCHK          ; Validate byte count
        .WORD   GENQIO          ; Issue write function
        .WORD   GENEND          ; End write processing next
                                ; word contains the address of
                                ; next jump after AST return
        .WORD   QIOEXM          ; Test for successful QIO
        .WORD   ALVBUF          ; Allocate verify buffer
        .WORD   GENQIO          ; Issue the Read QIO
        .WORD   ASTEXT          ; Exit AST
        .WORD   QIOEXM          ; Test for successful QIO
        .WORD   CMPBUF          ; Compare two buffers
        .WORD   DALBUF          ; Deallocate dynamic buffer
        .WORD   CLNUP           ; Success! Clean up, call PDDON
        .WORD   ASTEXT          ; Exit AST

; Bit functions
BITRD:
        .WORD   1               ; Bit read operation
        .WORD   IO.BRD          ; Unprotected read function
        .WORD   LCBINI          ; Initial register setup
```

```
        .WORD   VALBIT          ; Validate/byte align  bit pointer
        .WORD   CALCBY          ; Calculate number of bytes to read
        .WORD   RSRBYT          ; Set computed byte count in QIO
        .WORD   ALCBUF          ; Allocate dynamic buffer
        .WORD   GENQIO          ; Issue QIO read
        .WORD   GENEND          ; End issuing QIO read
        .WORD   QIOEXM          ; Determine successful QIO
        .WORD   RDMVBT          ; Move bits to buffer
        .WORD   DALBUF          ; Deallocate dynamic buffer
        .WORD   CLNUP           ; Success-cleanup, call PDDON
        .WORD   ASTEXT          ; End of generic function


BITWR:
        .WORD   1               ; Bit write entry table
        .WORD   IO.BWT          ; Bit write function
        .WORD   LCBINI          ; Initial register setup
        .WORD   VALBIT          ; Validate/byte align bit pointer
        .WORD   ALCBUF          ; Allocate dynamic buffer
        .WORD   CALCBY          ; Calc number of bytes to write
        .WORD   INIBIT          ; Bit write initialization
        .WORD   ALGNBT          ; Align bits for PC buffer address
        .WORD   ACNBYT          ; Convert register cnt to bytes
        .WORD   GENQIO          ; Issue QIO write
        .WORD   GENEND
        .WORD   QIOEXM          ; Determine successful QIO
        .WORD   DALBUF          ; Deallocate dynamic buffer
        .WORD   CLNUP           ; Success-cleanup, call PDDON
        .WORD   ASTEXT          ; End of generic function

BITWV:
        .WORD   2               ; Bit write verify entry table
        .WORD   IO.BWT          ; Bit write function
        .WORD   IO.BRD          ; Unprotected read function
        .WORD   LCBINI          ; Initial register setup
        .WORD   VALBIT          ; Validate/byte align bit pointer
        .WORD   ALCBUF          ; Allocate dynamic buffer
        .WORD   CALCBY          ; Calc number of bytes to write
        .WORD   INIBIT          ; Bit write initialization
        .WORD   ALGNBT          ; Align bits for PD buffer address
        .WORD   ACNBYT          ; Convert register count to bytes
        .WORD   GENQIO          ; Issue QIO write
        .WORD   GENEND          ; Leave generic routine
        .WORD   QIOEXM          ; Test for successful QIO
        .WORD   ALVBUF          ; Allocate dynamic buffer
        .WORD   RSRBYT          ; Set computed byte count in QIO
        .WORD   GENIO           ; Issue QIO read
        .WORD   ASTEXT          ; Exit first AST
        .WORD   QIOEXM          ; Test for successful QIO
        .WORD   RDMVBT          ; Move bits to buffer
```

```
        .WORD   BITCMP          ; Compare written with read bits
        .WORD   DALBUF          ; Deallocate dynamic buffer
        .WORD   DALBUF          ; Deallocate 2nd buffer
        .WORD   CLNUP           ; Success - Clean up,  call PDDON
        .WORD   ASTEXT          ; End of generic function

CBITRD:
        .WORD   1               ; Bit read operation
        .WORD   IO.BRD          ; Unprotected read function
        .WORD   LCBINI          ; Initial register setup
        .WORD   VALBIT          ; Validate/byte align  bit pointer
        .WORD   CALCBY          ; Calculate number of bytes to read
        .WORD   RSRBYT          ; Set computed byte count in QIO
        .WORD   ALCBUF          ; Allocate dynamic buffer
        .WORD   GENQIO          ; Issue QIO read
        .WORD   GENEND          ; End issuing QIO read
        .WORD   QIOEXM          ; Determine successful QIO
        .WORD   RDMVBT          ; Move bits to buffer
        .WORD   DALBUF          ; Deallocate dynamic buffer
        .WORD   CLNUP           ; Success-cleanup, call PDDON
        .WORD   ASTEXT          ; End of generic function


CBITWR:
        .WORD   1               ; Bit write entry table
        .WORD   IO.BWT          ; Bit write function
        .WORD   LCBINI          ; Initial register setup
        .WORD   VALBIT          ; Validate/byte align bit pointer
        .WORD   ALCBUF          ; Allocate dynamic buffer
        .WORD   CALCBY          ; Calc number of bytes to write
        .WORD   INIBIT          ; Bit write initialization
        .WORD   ALGNWD          ; Align bits for PD buffer address
        .WORD   ACNBYT          ; Convert register count to bytes
        .WORD   GENQIO          ; Issue QIO write
        .WORD   GENEND
        .WORD   QIOEXM          ; Determine successful QIO
        .WORD   DALBUF          ; Deallocate dynamic buffer
        .WORD   CLNUP           ; Success-cleanup, call PDDON
        .WORD   ASTEXT          ; End of generic function

BITWV:
        .WORD   2               ; Bit write verify entry table
        .WORD   IO.BWT          ; Bit write function
        .WORD   IO.BRD          ; Unprotected read function
        .WORD   LCBINI          ; Initial register setup
        .WORD   VALBIT          ; Validate/byte align bit pointer
        .WORD   ALCBUF          ; Allocate dynamic buffer
        .WORD   CALCBY          ; Calc number of bytes to write
        .WORD   INIBIT          ; Bit write initialization
        .WORD   ALGNWD          ; Align bits for PD buffer address
```

```
        .WORD   ACNBYT          ; Convert register count to bytes
        .WORD   GENQIO          ; Issue QIO write
        .WORD   GENEND          ; Leave generic routine
        .WORD   QIOEXM          ; Test for successful QIO
        .WORD   ALVBUF          ; Allocate dynamic buffer
        .WORD   RSRBYT          ; Set computed byte count in QIO
        .WORD   GENIO           ; Issue QIO read
        .WORD   ASTEXT          ; Exit first AST
        .WORD   QIOEXM          ; Test for successful QIO
        .WORD   RDMVBT          ; Move bits to buffer
        .WORD   BITCMP          ; Compare written with read bits
        .WORD   DALBUF          ; Deallocate dynamic buffer
        .WORD   DALBUF          ; Deallocate 2nd buffer
        .WORD   CLNUP           ; Success! Clean up, call $PDDON
        .WORD   ASTEXT          ; End of generic function

; Read status of PD

RDSTA:
        .WORD   1               ; One status QIO function
        .WORD   IO.DAG          ; Diagnostic read
        .WORD   LCBINI          ; Initial setup entry
        .WORD   SETSTS          ; Set up status QIO stuff
        .WORD   GENQIO          ; Issue QIO requested function
        .WORD   GENEND          ; End generic function, next
                                ; word contains the next jump
                                ; after entering the AST
        .WORD   QIOEXM          ; Test for successful QIO
        .WORD   CLNUP           ; Success clean up, call PDDON
        .WORD   ASTEXT          ; Exit AST


START:
        .WORD   1               ; One status QIO function
        .WORD   IO.STR          ; Start the device
        .WORD   LCBINI          ; Initial setup entry
        .WORD   STRSTP          ; Set up a start/stop operation
        .WORD   GENQIO          ; Issue QIO requested function
        .WORD   GENEND          ; End generic function, next
                                ; word contains the next jump
                                ; after entering the AST
        .WORD   QIOEXM          ; Test for successful QIO
        .WORD   CLNUP           ; Success clean up, call PDDON
        .WORD   ASTEXT          ; Exit AST

STOP:
        .WORD   1               ; One status QIO function
        .WORD   IO.STP          ; Stop the device
        .WORD   LCBINI          ; Initial setup entry
        .WORD   STRSTP          ; Set up a start/stop operation
        .WORD   GENQIO          ; Issue QIO requested function
```

```
        .WORD   GENEND          ; End generic function, next
                                ; word contains the next jump
                                ; after entering the AST
        .WORD   QIOEXM          ; Test for successful QIO
        .WORD   CLNUP           ; Success clean up, call PDDON
        .WORD   ASTEXT          ; Exit AST

; Fill in necessary entry points to perform the requested
; generic functions for the particular devices you're
; supporting.

STOP:
UPREQ:
UPEND:
DNREQ:
DNEND:
ASTOP:
NOOPEX:
        .WORD   LCBINI
        .WORD   NOFUNC
        .WORD   GENEND                  ; End generic function
```

```
.xxGEN:

;-----------------------------------------------------------------
;
; $GENFC uses the entry point in manufacturer/model tables to
; determine the generic function to be processed and returns the
; entry point function branch table
;
; Input:
;       R0 points to entry in manufacturer table
;       R5 points to LCB
;
; Output:
;       R0 points to function branch table for requested function
;       R1 points to QIO function to be used next.      (R1 < R0)
;       R5 points to LCB
;
;
; $GENFC uses the entry point in manufacturer/model to determine
; the generic function to be processed and returns the entry point
; function branch table
;

        SAVRG$  <R5,R4,R3,R2,R1,R0> ; Save registers
        MOV     #xxTBL,R0           ; R0 points to Manf/model table
        CALL    $GENFC              ; Determine function branch
                                    ; table

        CMP     #-1,R0              ; Function error?
        BEQ     10$                 ; Yes, end generic request
        TST     R0                  ; If zero, no-op function
        BNE     5$                  ; Process specified function
        MOV     #NOPEX,R0           ; NO-OP function for device
5$:
        JMP     @(R0)+              ; Jump to first sub function
10$:
        MOV     #SE.FNR,C.SRVS(R5) ; Generic function error code
        JMP     GENERR              ; End generic processing for

;                   Generic Function Entry Points

;-----------------------------------------------------------------

LCBINI:
;
; Initialize the fields in the LCB and others so that the
; processing may begin.

        MOV     #GENERR, C.SAVE+30.(R5) ; GENERR is the default
                                        ; error routine for now
```

```
        CLR     C.SAVE+16.(R5)  ; Clear allocate buffer lengths
        CLR     C.SAVE+18.(R5)  ; to indicate that nothing has
        CLR     C.SAVE+20.(R5)  ; been allocated yet

        MOV     C.MADR(R5), R3  ; R3 points to Start generic request
                                ; message
        MOV     R1, C.SAVE(R5)  ; Address of current QIO func
        MOV     R.GCNT(R3), C.SAVE+4(R5) ; Save Count of data
                                ; requested
        CLR     C.SAVE+8.(R5)   ; Clear subfunction
        MOV     R3,C.SAVE+2(R5) ; Copy beginning of message data
        ADD     #R.GDAT,C.SAVE+2(R5) ; Address of data buffer
        JMP     @(R0)+          ; Goto next entry in table

;-------------------------------------------------------------

ALABUF:

; For model MD.XXC, allocate memory within task image for
; extended address.  This is necessary since the driver assumes
; that the address buffer is in the task space.  Since it is in
; the mapped message buffer it is not available to the driver.

        SAVRG$  <R0>            ; Save registers, $RQCB destroys
                                ; R0, R1, & R2
        MOV     #XTL.ADR,R1     ; Set buffer size
        MOV     #F$REHD,R0      ; R0 points to free memory listhead
        CALL    $RQCB           ; Allocate new block
        BCC     1$              ; Clear? - it made it

        ; Error allocating buffer
        MOV     #SE.RSC,C.SRVS(R5) ; Resource error
        CLR     C.IOSB(R5)      ; Clear 1st word of IOSB
        CLR     C.IOSB+2(R5)    ; Clear byte count in IOSB
        RESRG$  <R0>            ; Restore registers
        JMP     @C.SAVE+30.(R5) ; Go to error routine
1$:
        MOV     R0,C.SAVE+6(R5) ; Address of address buffer
        MOV     R1,C.SAVE+16.(R5) ; Save length for deallocation
        MOV     C.MADR(R5),R1   ; R1 points to Start of Generic data
        ADD     #R.GADR,R1      ; Address of extended address
        .REPT   <XTLADR/2>      ; to move
        MOV     (R1)+,(R0)+     ; Move the extended address into
        .ENDR                   ; task image space
        RESRG$  <R0>            ; Restore registers
        JMP     @(R0)+          ; Goto next entry in table

;-------------------------------------------------------------
;
CNVBYT:
;
```

```
; Switch register count to byte count for R/W register

        ASL     C.SAVE+4(R5)    ; Shift # words to bytes of data
        JMP     @(R0)+          ; Go to next entry in table

;-------------------------------------------------------------
;
BYTCHK:
;
; Compare byte count requested with current byte count for
; message buffer.

        MOV     C.MADR(R5),R1   ; R1 points to Start of Generic data
        MOV     M.DLEN(R1),R1   ; R1 points to previous returned byte
                                ; count
        SUB     #R.GDAT,R1      ; Actual data bytes
        BLE     50$             ; Need more than zero

        CMP     R1, #R.GLDT     ; is it bigger than the max i/o
        BLE     20$             ; transaction for this device?
        MOV     #R.GLDT, R1     ; Use the smaller of the two.
20$:
        CMP     R1,C.SAVE+4(R5) ; Compare to requested
        BLT     30$             ; Need  >= 0
        JMP     @(R0)+          ; Go to next table entry
30$:
        MOV     #SE.CNT, C.SRVS(R5) ; Bad count requested.
        JMP     @C.SAVE+30.(R5) ; Go to error routine


;-------------------------------------------------------------

ALCBUF:

; Allocate dynamic buffer area
        SAVRG$  <R0>            ; Save registers

        MOV     G$CBYN, R1
        SUB     #R.GDAT, R1     ; make actual
        CMP     R1, #R.KGLDT    ; Compare buffer size with
                                ; allowed for QIO maximum
        BLE     10$             ; Take the smaller of the two
        MOV     #R.KGLDT,R1     ; Set buffer size
10$:
        MOV     #F$REHD,R0      ; R0 points to free memory listhead
        CALL    $RQCB           ; Allocate new block
        BCS     30$             ; Set? - it didn't allocate

        MOV     C.SAVE+2.(R5), C.SAVE+12.(R5) ; Save old
                                ; buffer address
```

```
        MOV     C.SAVE+18.(R5),C.SAVE+20.(R5) ; Save old
                                              ; buffer length
        MOV     R0,C.SAVE+2(R5) ; Move in new buffer address
        MOV     R1,C.SAVE+18.(R5) ; Set actual length
        RESRG$  <R0>            ; Restore registers
        JMP     @(R0)+          ; Go to next entry

30$:    ; Error allocating buffer
        MOV     #SE.RSC,C.SRVS(R5)      ; Resource error
        RESRG$  <R0>                    ; Restore registers
        JMP     @C.SAVE+30(R5)  ; Nested AST level error

;-------------------------------------------------------------

VALBIT:

; Validation and possible word points to byte adjustment of bit pointer
; Device XXC needs to be word aligned others byte

        MOV     C.MADR(R5),R2   ; R2 points to Start generic data
        CMPB    #16.,R.GBOF(R2) ; Check for valid bit offset
        BGT     10$             ; Less than okay

        MOV     #SE.ADR, C.SRVS(R5); Generic error, reject
                                ; function because of address
        JMP     @C.SAVE+30.(R5) ; Flag error and return
10$:
        CMP     C.MODL(R5),#MD.XXC ; Is this a XXC function?
        BEQ     20$             ; Yes, leave as word aligned

        CMPB    #8.,(R2)        ; Pointing to second byte
        BGT     20$             ; Fits in first byte
        SUB     #8.,(R2)        ; Point to bit within byte
        INC     C.SAVE+6(R5)    ; Move addr up 1 byte
20$:
        JMP     @(R0)+          ; Go to next entry in table

;-------------------------------------------------------------

INIBIT:

; Initial register setup for bit alignment operations
; When exiting, R2 = bit offset negated for right shift

        MOV     C.MADR(R5),R3   ; R3 points to Start generic data
        MOV     R.GBOF(R3),R2   ; Minus bit offset
        NEG     R2              ; Shift to the right
        ADD     R.GBOF(R3),C.SAVE+4(R5) ; Add due to bit offset
        MOV -   C.SAVE+2(R5), R4 ; Address of buffer
        MOV     #-16.,C.SAVE+28.(R5) ; Possible shifts plus one
        ADD     R.GADR+30.(R3),C.SAVE+28.(R5) ; Setup alignment
```

```
                                        ; number
        JMP     @(R0)+                  ; Go to next table entry

;-------------------------------------------------------------

ALGNBT:

; Bit alignment for bit write operations
; R2 = bit offset into word (negated for right shift operation)

        SAVRG$  <R0>                    ; Save registers
        CLR     R1                      ; Clear low order second register
        MOV     C.MADR(R5),R3           ; R3 points to Beginning of message data
        ADD     #R.GDAT,R3              ; Beginning of message data
10$:
        MOV      C.SAVE+6(R5),(R4)+ ; Move PC address to buffer
        INC     C.SAVE+6(R5)            ; Point to next word in PC
        MOV     (R3)+,R0                ; R1 points to Bit data
        SAVRG$  <R3>                    ; Save pointer to data
        MOV     C.MADR(R5),R3           ; R3 points to Beginning of message data
        ASHC    C.SAVE+28.(R3),R0 ; Shift double word needed bits
        MOV     R1,(R4)                 ; Store set bits in out buffer
        MOV     (R4)+,2(R4)             ; Set up clear word 2 words away
        MOV     C.SAVE+6(R5),(R4)+ ; Move next PC address in buffer
        INC     C.SAVE+6(R5)            ; Ready for next PC address
        COM     (R4)                    ; Reset mask for bits cleared
        CMP     C.SAVE+4(R5),R.GCNT(R3) ; Shifting first word
        BLT     30$                     ; No, into bit operation

; We are on first word, save the bits not requesting change

        SAVRG$  <R0>                    ; Save for next shift
        MOV     #-1,R0                  ; Set mask to all ones
        ASH     R.GBOF(R3),R0 ; Shift bit pointer amount
        COM     R0                      ; Save mask bits set to 1s
        BIC     R0,(R4)                 ; Save low order bits
        RESRG$  <R0>                    ; Restore for next shift
30$:
        ASHC    R2,R0                   ; Setup for next shift
        MOVB    (R4),R0                 ; Swap low with high order next word
        MOVB    -3(R4),(R4)             ; Move high order to next word
        MOVB    R0,-3(R4)               ; Move low clear to high prev. byte
        ADD     #2,R4                   ; Point to next address
        SUB     #16.,C.SAVE+4(R5) ; Decrease number of bits left
        RESRG$  <R3>                    ; Restore pointer to data
        BGT     10$                     ; not done yet

; We are on last word
; Save bits in high order that did not request updating

        MOV     #-1,R0                  ; Set up save mask
```

```
        MOV     #16.,R1             ; Number of possible shifts +1
        ADD     C.SAVE+4(R5),R1     ; Subtract number shifted
        ASH     R1,R0               ; Shift mask for low order byte
        MOV     R0,R1               ; R1 for high order
        SWAB    R1                  ; Save mask for high byte
        SUB     #6,R4               ; Point back to set word
        BICB    R0,(R4)             ; Save clear bits
        BICB    R0,1(R4)            ; Save set bits
        BICB    R1,4(R4)            ; Save high order set bits
        BICB    R1,5(R4)            ; Save high order clear bits
        MOV     C.SAVE+14.(R5),C.SAVE+4(R5) ; Store number of
                                    ; bytes to write
        MOV     C.MADR(R5),R3       ; R3 points to Beginning of message data
        MOV     R.GADR(R3),C.SAVE+6(R5) ; Reset PC address
        RESRG$  <R0>                ; Restore register
        JMP     @(R0)+              ; Go to next entry in table

;-------------------------------------------------------------

ALGNWD:

; Bit alignment within word for XXC write bit operations
; R2 = bit offset into word (negated for right shift operation)

        SAVRG$  <R0>                ; Save registers
        CLR     R1                  ; Clear low order second register
        MOV     C.MADR(R5),R3       ; R3 points to Beginning of message data
        ADD     #R.GDAT,R3          ; Beginning of message data

        CMP     #16.,C.SAVE+4(R5)   ; Test for one word byte count
        BGE     10$                 ; If within one word okay
        MOV     #SE.CNT,C.SRVS(R5)  ; Move in invalid count error
                                    ; to srvc sts
        RESRG$  <R0>                ; Restore registers
        JMP     @C.SAVE+30(R5)      ; Error route out

10$:
        MOV     (R3)+,R0            ; R1 points to Bit data
        MOV     C.MADR(R5),R3       ; R3 points to Beginning of message data
        ASHC    C.SAVE+28.(R5),R0   ;Shift double word needed bits
        MOV     R1,(R4)             ; Store set bits in out buffer
        MOV     (R4)+,(R4)          ; Set up clear word
        COM     (R4)+               ; Reset mask for bits cleared

; We are on first word, save the bits not requesting change
; First and only for now

        MOV     #-1,R0              ; Set mask to all ones
        ASH     R.GBOF(R3),R0 ;  Shift bit pointer amount
        COM     R0                  ; Saved mask bits zeroed
        BIC     R0,-(R4)            ; Save low order bits
```

```
30$:

; We are on last word
; Save bits in high order that did not request updating

        MOV     #-1,R0              ; Set up save mask
        ASH     C.SAVE+4(R5),R0     ; Shift mask
        BIC     R0,(R4)             ; Save clear bits
        BIC     R0,-(R4)            ; Save set bits

; Force a one word entry( ACP accepts only one word QIO)

        MOV     #4,C.SAVE+4(R5)     ; Store number bytes to write
40$:    RESRG$  <R0>                ; Restore register
        JMP     @(R0)+              ; Go to next entry in table

;-------------------------------------------------------------

CALCBY:

; Calculate number of bytes from bits sent down for QIO operation

        MOV     C.MADR(R5),R3       ; R3 points to Beginning of message data
        MOV     R.GBOF(R3),R1       ; Bit pointer into byte
        ADD     C.SAVE+4(R5),R1     ; Number of bits working with
        ADD     #7, R1              ; Force a round up to the next byte
        ASH     #-3,R1              ; Divide by 8
10$:
        CMP     M.CODE(R3),#MG.RED  ; Bit read?
        BEQ     20$                 ; Yes, no addr in data
        ASL     R1                  ; Double, due to masking space
20$:
        CMP     C.MODL(R5),#MD.XXC  ; Is this XXC function?
        BNE     30$                 ; Go on
        ADD     #1,R1               ; Inc to next byte
        BIC     #1,R1               ; Force even

30$:    MOV     R1,C.SAVE+14.(R5)   ; Number of bytes to write
        JMP     @(R0)+              ; Go to next table entry

;-------------------------------------------------------------

RSRBYT:

; Place the calculated byte count into QIO parameters

        MOV     C.SAVE+14.(R5),C.SAVE+4(R5) ; Calculated number of
                                            ; parameters sent
        JMP     @(R0)+                  ; Go to next table entry

;-------------------------------------------------------------
```

```
SETSTS:

  Set up device status operation

        MOV     #10.,C.SAVE+4(R5) ; 10 bytes expected
        CMP     C.MODL(R5),#MD.XXC ; Is this a XXC request
        BNE     10$               ; Bypass byte count change
        MOV     #18.,C.SAVE+4(R5) ; 18 bytes for XXC
        MOV     C.SAVE+8.(R5),C.SAVE(R5) ; Move in status
                                  ; function
        BR      20$               ; Need subfunction yet
10$:
        MOV     C.SAVE(R5),C.SAVE+8.(R5) ; Get address of
                                  ;functions
20$:
        ADD     #2,C.SAVE+8.(R5); Point to subfunction
        JMP     @(R0)+            ; Go to next table entry

;-----------------------------------------------------------------

STRSTP:

; Set up start stop operation

        CLR     C.SAVE+4(R5)      ; There is no data to betransferred
        JMP     @(R0)+            ; Go to next table entry

  ----------------------------------------------------------------

GENQIO:

; QIO's parameters were initially set up in the LCBINI function
; routine parameters are stored in C.SAVE(1) thru C.SAVE(8) for
; easy retrieval in case of timeouts or block check retries.
;
;  The QIO parameters for the QIO are

        MOV     C.LAB(R5),R4      ; R4 points to LAB for this device
        MOV     L.LUN(R4),R4      ; R4 points to Logical unit number
        MOV     R5,R3             ; Copy LCB beginning
        ADD     #C.SAVE,R3        ; R3 points to Beginning of saved data
        MOV     #GENAST, R2 ; Get address of completion AST
        MOV     R0,C.SAVE+10.(R5) ; Save next entry of function
                                  ;   br tbl
        ADD     #2,C.SAVE+10.(R5) ; Point to next entry beyond...

 QIO$S @(R3),R4,,,R5,R2,<2(R3),4(R3),C.STN(R5),6(R3),@8.(R3),TMOUT>

        MOV     C.MADR(R5),R3     ; R3 points to Beginning of generic data
        MOV     #DSW,C.SRVS(R5) ; Generic status message
        BMI     50$               ; Take error route if directive
```

```
                                        ; error
        JMP     @(R0)+                  ; Go to next table entry point
50$:
        JMP     @C.SAVE+30.(R5) ; Send error and exit

;-------------------------------------------------------------

GENAST:

; Generic service AST reentry point and recovery routine

        ASDIM$                          ; Save registers R5-R0,
                                        ; R5 points to LAB of operation, and
                                        ; message buffer pointed to
                                        ; by C.MPNT(R5) and C.MLEN(R5)
                                        ; is remapped and the address of
                                        ; message header is placed in
                                        ; C.MADR(R5)

        MOV     #ASTERR, C.SAVE+30.(R5) ; Default error routine
                                        ; is now AST type

        ; Determine if there has been a time out or block check

        CMPB    C.IOSB(R5),#IE.TMO      ; Test for time out condition
        BEQ     10$                     ; Yes, dec count and retry
        CMPB    C.IOSB(R5),#IE.BCC      ; Test block check condition
        BNE     15$                     ; No, determine status and condition
10$:
        DEC     C.TRY(R5)               ; Number of retries
        BLE     15$                     ; Done trying - continue

        ; Re - issue QIO

        MOV     C.LAB(R5),R4            ; R4 points to LAB for this device
        MOV     L.LUN(R4),R4            ; R4 points to Logical unit number
        MOV     R5,R3                   ; Copy LCB beginning
        ADD     #C.SAVE,R3              ; R3 points to Beginning of saved data
        MOV     #GENAST, R2

    QIO$S @(R3),R4,,,R5,R2,<2(R3),4(R3),C.STN(R5),6(R3),@8.(R3),TMOUT>

        MOV     C.MADR(R5),R3   ; R3 points to Beginning of generic data
        MOV     $DSW,C.SRVS(R5) ; Return $DSW error code

        BMI     20$                     ; Finished generic request
15$:
        MOV     C.SAVE+10.(R5),R0 ; Restore pointer to function
                                        ; branch table
        JMP     @(R0)+                  ; Go to next function branch entry
```

```
; DSW error

 )$:
        CLR     R1                      ; For other than read functions
        JMP     @C.SAVE+30.(R5) ; take exit route

;-----------------------------------------------------------------

@IOEXM:

; Determine RSX i/o error then send the generic error code, if
; successful then continue with next step in function branch
; table

20$:
        CLR     C.SRVS(R5)              ; Generic error message area
        MOV     #IOERR,R1               ; R1 points to i/o status error table
        MOV     #IOERCT,R2              ; R2 points to Size of error table
29$:
        CMPB    C.IOSB(R5),(R1) ; Determine i/o status
        BEQ     30$                     ; go to insert generic function
        ADD     #2,R1                   ; Try next entry
        SOB     R2,29$                  ; Do until end of table
        MOV     #SE.INT,C.SRVS(R5) ; Function error returned
        BR      35$                     ;
30$:
        MOVB    1(R1),R1                ; Byte status error
        MOV     R1,C.SRVS(R5)   ; Insert generic status
        CMP     R1,SE.SUC               ; Was it successful?
        BNE     35$
        JMP     @(R0)+
35$:
        CLR     R1                      ; Error - clear byte cnt
        JMP     @C.SAVE+30.(R5) ; End generic function

;-----------------------------------------------------------------

RDMVBT:

; Align bits to send back upstairs for requested read bit
; operation.  Send bit string format of only bits request
; beginning with bit offset into word

        SAVRG$  <R0>                    ; Save registers
        MOV     C.MADR(R5),R3   ; R3 points to Beginning of message data
        MOVB    R.GBOF(R3),R2   ; Bit pointer within word
        NEG     R2                      ; For right shifts
        MOV     C.SAVE+4(R5),C.SAVE+14.(R5) ; Copy bytes read in
        MOV     C.SAVE+2(R5),R3 ; Allocated buffer read into
        MOV     C.SAVE+12.(R5),R4; Return buffer
10$:
```

```
        MOV     (R3)+,R1            ; First word in low order register
        MOV     (R3),R0             ; Move into high order register
        ASHC    R2,R0               ; Shift to the right
        MOV     R1,(R4)+            ; Move low order to message buffer
        SUB     #2,C.SAVE+14.(R5)   ; Decrease bytes left to shift
        BGT     10$                 ; Not done, shift again

        MOV     #^C0,R0             ; Set up mask
        MOV     C.MADR(R5),R3       ; R3 points to Beginning of message data
        MOV     R.GCNT(R3),R2       ; Save bit count
        BIC     #^C15.,R2           ; Mask out all but word
        BEQ     20$
        ASH     R2,R0               ; Shift masked bits
        BIC     R0,-(R4)            ; Save bits not requested
20$:
        RESRG$  <R0>                ; Restore registers
        JMP     @(R0)+

;-------------------------------------------------------------

BITCMP:

; Bit comparison for write/verify bit operation

        SAVRG$  <R0>                ; Save registers
        MOV     C.MADR(R5),R3       ; R3 points to Beginning of message data
        MOV     R.GCNT(R3),R2       ; Number of bits written
        ADD     #R.GDAT,R3          ; R3 points to Start of input bits
        MOV     C.SAVE+12.(R5),R4   ; R4 points to Bits read in from PC
10$:    SUB     #8.,R2              ; Decrease number of bits left
        BGE     20$                 ; More bytes to go

; We are on the last byte to compare

        MOV     #-1,R0              ; Set up mask
        ASH     R2,R0               ; To the left
        BICB    R0,(R3)             ; Clear input
        BICB    R0,(R4)             ; Clear read in
20$:    CMPB    (R3),(R4)           ; Compare current byte
        BNE     30$                 ; Bits not the same?
        TST     R2                  ; Done?
        BGT     10$                 ; No, go to next byte

        RESRG$  <R0>                ; Restore register
        JMP     @(R0)+              ; Go to next table entry

30$:    MOV     #SE.VFY,C.SRVS(R5)  ; Verification error
        JMP     @C.SAVE+30.(R5)     ; End of generic function

;-------------------------------------------------------------
```

```
CMPBUF:

  Routine to compare buffer written and buffer read back

        MOV     C.SAVE+4(R5),R1 ; R1 points to Number of bytes to compare
        MOV     C.MADR(R5),R3   ; R3 points to Start generic data
        ADD     #R.GDAT,R3      ; Write buffer
        MOV     C.SAVE+2(R5),R2 ; R2 points to Beginning of read buffer
10$:
        CMPB    (R2)+,(R3)+     ; Test current byte and inc
        BNE     30$             ; Error in data
        SOB     R1,10$          ; Move thru buffer areas
        JMP     @(R0)+          ; Go to next table entry

30$:    MOV     #SE.VFY,C.SRVS(R5)      ; Verification error
        MOV     C.MADR(R5),R3   ; R3 points to Start generic data
        MOV     #SE.VFY,R.GSTS(R3) ; Generic error
        JMP     @C.SAVE+30.(R5) ; End generic function

;---------------------------------------------------------------

DALBUF:

; Deallocate temporary work buffer


        TST     C.SAVE+18.(R5)  ; Get length of dealloc buffer
        BEQ     10$             ; Nothing to deallocate

        SAVRG$  <R0,R1>         ; Save registers
        MOV     #F$REHD,R0      ; Address of free memory
                                ; listhead
        MOV     C.SAVE+18.(R5),R1 ; Size of block to be released
        MOV     C.SAVE+2.(R5), R2 ; Address of block released
        CALL    $RLCB           ; Release it
        MOV     C.SAVE+12.(R5),C.SAVE+2.(R5) ; Restore old
                                ;       buffer area
        MOV     C.SAVE+20.(R5),C.SAVE+18.(R5) ; Restore old
                                ; buffer length
        CLR     C.SAVE+20.(R5)  ; Mark second buffer deallocated
        RESRG$  <R1,R0>         ; Restore registers

10$:
        JMP     @(R0)+                  ; Go to next entry

;---------------------------------------------------------------

DALXTA:

; Deallocate memory used for storing address buffers within
; task image space.
```

```
        TST     C.SAVE+16.(R5)  ; Was memory allocated for a
        BEQ     20$             ; extended address function ?

        SAVRG$  <R0,R1>         ; Save registers
        MOV     #F$REHD,R0      ; Address of free memory
                                ; listhead
        MOV     C.SAVE+16.(R5),R1 ; Size of block to be released
        MOV     C.SAVE+6.(R5),R2; Address of block released
        CALL    $RLCB           ; Release it
        CLR     C.SAVE+16.(R5)  ; Mark it deallocated
        RESRG$  <R1,R0>         ; Restore registers

20$:
        JMP     @(R0)+          ; Go to next entry

;-------------------------------------------------------------

CLNUP:

; Successfully completed requested generic function
; Set up status returns and registers then call PDDON

        MOV     C.MADR(R5),R3   ; R3 points to Beginning of generic data
        MOV     C.LAB(R5),R4    ; R4 points to Associated LAB
        MOV     #SS.SUC,C.SRVS(R5) ; Generic status
        MOV     C.SAVE+4(R5),R1 ; Save byte count
        CALL    $PDDON          ; Perform cleanup operations
        JMP     @(R0)+          ; Finish exit AST

;-------------------------------------------------------------

NOFUNC:

; NOP function routine for devices not supporting all generic
; functions

        CLR     R1              ; Zero number of bytes
        MOV     #SS.SUC,C.SRVS(R5) ; Generic success message
        CALL    $PDDON
        JMP     @(R0)+          ; Generic operation doesn't
                                ; for current device

;-------------------------------------------------------------

GENERR:
        MOV     #1, R0
        BR      5$
ASTERR:
        CLR     R0
;
```

```
; Error, clean up routine. Clean up any allocate buffers. Clear any
  data being sent back and either return or exit AST depending
  on which entry point taken
;
5$:
        MOV     C.MADR(R5),R3   ; R3 points to Beginning of generic data
        ADD     #R.GDAT,R3      ; Point to address of data

        SAVRG$  <R0>
        MOV     #F$REHD,R0

;  Deallocate primary buffer

        TST     C.SAVE+18.(R5)  ; Was memory allocated for the
        BEQ     10$             ; primary buffer ?

        MOV     C.SAVE+18.(R5),R1 ; Size of block to be released
        MOV     C.SAVE+2.(R5),R2 ; Address of block released
        CALL    $RLCB           ; Release it

;  Deallocate secondary buffer

10$:
        TST     C.SAVE+20.(R5)  ; Was memory allocated for the
        BEQ     20$             ; secondary buffer ?

        MOV     C.SAVE+20.(R5),R1 ; Size of block to be released
        MOV     C.SAVE+12.(R5),R2 ; Address of block released
        CALL    $RLCB           ; Release it


; Deallocate extended address buffer

20$:
        TST     C.SAVE+16.(R5)  ; Was memory allocated for a
        BEQ     30$             ; PLC3 extended address function ?

        MOV     C.SAVE+16.(R5),R1 ; Size of block to be released
        MOV     C.SAVE+6.(R5),R2 ; Address of block released
        CALL    $RLCB           ; Release it

30$:
        RESRG$  <R0>

        CLR     R1              ; Zero byte count
        CALL    $PDDON          ; - so function is bypassed
        TST     R0              ; Determine how we get out of here
        BEQ     50$
        JMP     GENEND          ; Return to caller
50$:
        JMP     ASTEXT          ; Return from AST
```

```
;-------------------------------------------------------------------

GENEND:

; Exit at non-completion AST level

        RESRG$  <R0,R1,R2,R3,R4,R5> ; Restore registers
        RETURN                      ; Return to caller

;-------------------------------------------------------------------

ASTEXT:

; Exit intermediate AST

        RESRG$  <R0,R1,R2,R3,R4,R5> ; Restore registers
        ASTX$S                      ; Exit AST

        .DSABL LSB
```

APPENDIX D

ADDING NEW DEVICE SUPPORT

D.1 Overview

The information presented in this Appendix is useful to those individuals who are adding new device support.

D.2 Useful Reading Material

The following publication may be useful when planning the addition of new device support:

- SHOP FLOOR GATEWAY Installation Guide/Release Notes
- User's Guide to the Allen-Bradley Data Highway Network Communication Software (13/NET)

  (1770-843, available from Allen-Bradley; useful if the user is writing a DIM for unsupported Allen-Bradley devices, or another utility)

D.3 Hardware / Software Environment

The GATEWAY system is implemented on PDP-11 series processors, each containing a minimum of 512K bytes of memory. A maximum of four gateways are supported. The processors are connected to a VAX via a DECnet link and to programmable device networks via asynchronous serial lines.

The GATEWAY system uses the RSX-11S operating system. This operating system features minimal overhead while providing a real-time multitasking environment.

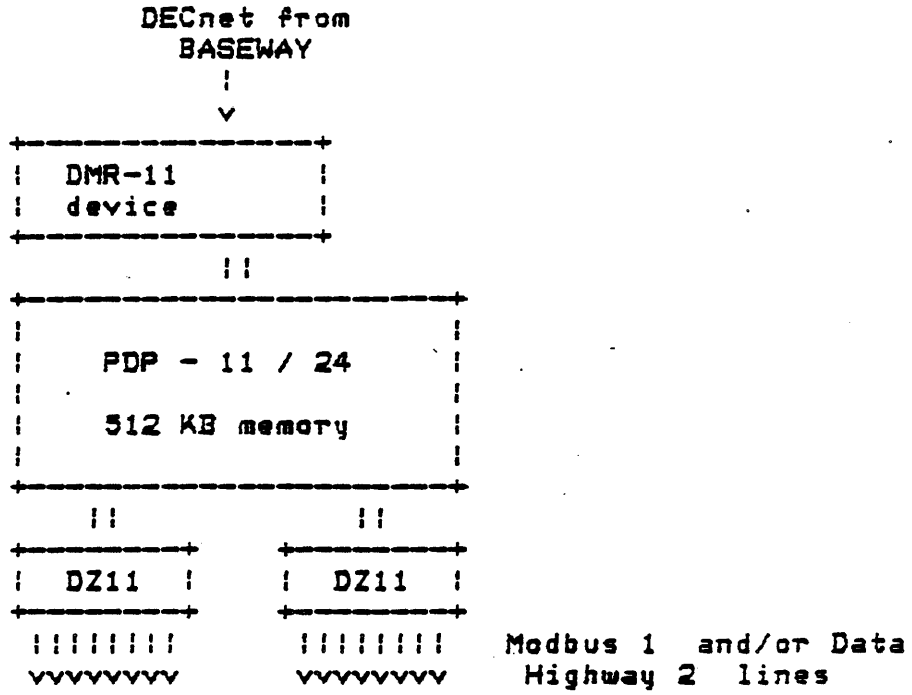


Figure 11. BASEWAY Network

1 Modbus TM is a registered trademark of Gould Incorporated, Modicon Division.
2 Data Highway TM is a registered trademark of Allen-Bradley Corporation.

D.4 SHOP FLOOR GATEWAY Tasks

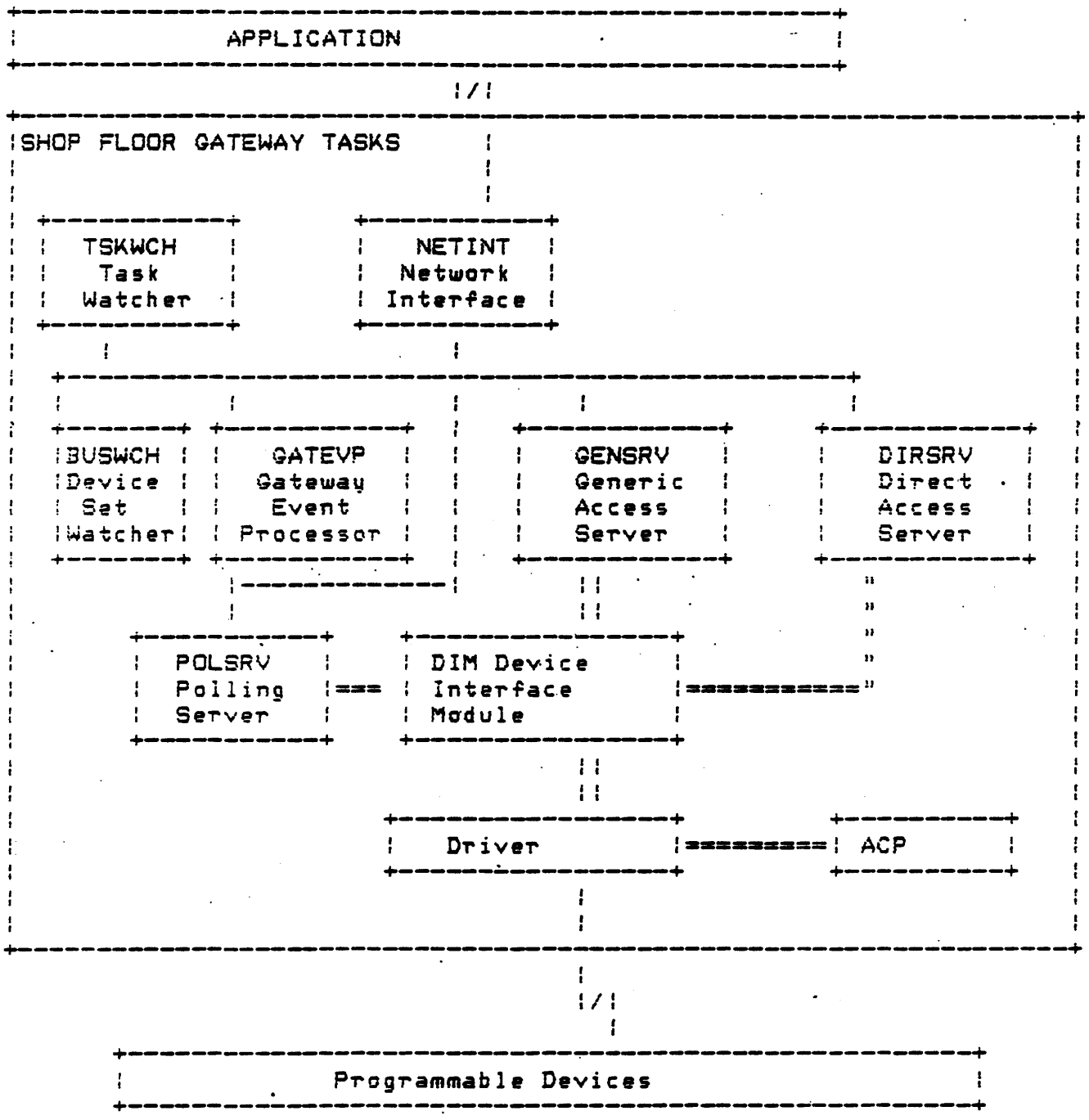


Figure 12. SHOP FLOOR GATEWAY Tasks

D.5 GATEWAY Initialization

A System Startup is performed when the GATEWAY system is first powered on, and thereafter whenever a complete GATEWAY restart is requested. Initially, DECnet downline loads the GATEWAY system and programs.

After the operating system is initialized and certain tasks that perform system and application initialization have been downloaded and completed, a copy of the Task Watcher task is downline task-loaded from BASEWAY. The Task Watcher task then requests the network interface, creates the intertask mailboxes, and starts the other tasks running.

Initialization of the GATEWAY tables is performed after everything is in readiness. The GATEWAY Initializer on the application traverses the required databases and sends messages containing the polling parameters to the GATEWAY Event Processor. GATEVP builds the memory-resident polling tables from these messages and enables the Polling Interface.

D.5.1 Network Interface (NETINT)

The Network Interface task allows tasks to communicate regardless of DECnet limitations or functionality. In addition, it performs the following actions:

- except for certain control messages, routes incoming interprocess messages to the proper destination transparently;
- maintains queues for outgoing messages;
- optimizes message traffic over DECnet;
- communicates directly with other Network Interface tasks on other processors as required.

D.5.2 GATEWAY Event Processor (GATEVP)

The GATEWAY Event Processor subsystem is responsible for controlling the operation of the GATEWAY system. The GATEVP task resides in every GATEWAY system, and is controlled and monitored by the TSKWCH task.

On GATEWAY initialization, the GATEWAY Event Processor receives configuration messages from BASEWAY which are used to set up VDR database. This database is used by the Polling Interface to determine the polling requirements and status of the system.

Its activities include:

- sending raw PD data to BASEWAY after qualification and preprocessing, depending on register definition;
- accepting messages from BASEWAY to load PD definitions in VDR;
- returning a status report to BASEWAY on command;
- interacting with the Network Interface and various system management functions on the SFG;

5.2.1 PD Data Processing -

The GATEVP task performs PD register qualification and any preprocessing of data before the message reaches BASEWAY. This includes:

- analyzing whether a status bit or data changed before sending it to BASEWAY.
- determining whether to process trigger buffers;
- converting PD register data from BCD to binary numbers;

Data to be sent is first formatted into polled data messages. Next, depending on the type of register being processed, one or more data packets are packed into the message for transmission. The process which GATEVP sends a data message to is part of that data's definition (the default is DATA_PROC). One data packet is defined for each each status or maintenance bit, trigger buffer, or data count register. Each polled data message has a header containing the equipment logical ID and a time-date stamp. Each data packet is identified with data identifier code.

D.5.3 GATEWAY Task Watcher (TSKWCH)

The Task Watcher task:

- monitors the status of each task in the GATEWAY system, and informs the application when problems occur;
- handles system startup and shutdown.

The Task Watcher task runs continuously on the GATEWAY.

D.5.4 Device Set Watcher (BUSWCH)

The Device Set Watcher task monitors and controls the accessibility of device sets for the GATEWAY. It is responsible for initially determining what device sets are currently attached to the GATEWAY. It is also responsible for synchronizing the access server activity with these events.

D.5.5 Direct Access Server (DIRSRV)

The Direct Access Server task provides a direct driver interface to a programmable device, bypassing the protocol-handling features of the PD support Ancillary Control Processes (ACPs). In addition, the task

- does not add device-specific protocol to messages sent to programmable devices (allowing user application programs on BASEWAY to do protocol for special situations);
- performs simple transactions on the device networks.

The Direct Access Server is useful when a vendor-supplied software program exists to do downline loads, upline dumps, etc. For example, in Modbus TM applications the standard Modicon software on BASEWAY does all of the protocol and retry processing and uses the Direct Access Server to process one Modbus TM packet at a time.

D.5.6 Generic Access Server (GENSRV)

The programmable device Generic Access Server is responsible for performing any generic i/o to a programmable device. It performs register read/write functions, coil read/write, unprotected memory access, etc. Thus, application programs running on BASEWAY may read and write to programmable devices without regard to protocols, unique device architectures, addressing schemes, and other such matters. The Generic Access Server task:

- accesses devices via simple requests such as "read register," "write coils," etc.;
- handles protocol, retries, etc., in conjunction with the device support ACPs.

The request and reply messages have the same general format regardless of the make or model of device.

D.5.7 Polling Server (POLSRV)

The Programmable Device Polling subsystem is implemented by the POLSRV task. It is responsible for systematically "polling" each PD and informing the rest of the system whenever polled data changes. The Polling Server task is driven by tables which are initially built on the application and are loaded into tables in the GATEWAY by the Event Processor task. Polling is done for each device specified in the table, with a given set or sets of data definitions being polled at specified intervals. In general, data from the scans is sent back to BASEWAY only if a point value changes.

The POLSRV task is also a highly data-driven program. It scans the data structures in the Virtual Data Region, pausing at specific PD polling set blocks to poll selected PD data. Whether a PD is polled depends on a number of flags and status bits in the VDR data structures, and whether polling a PD generates any further processing depends on the previous contents of the registers and the data type.

D.5.8 Adding a New Device to a GATEWAY

The following steps may be used as a guide to adding a new device to the SHOP FLOOR GATEWAY:

1. Decide if a terminal driver can be used. If it cannot, you must write a device driver and ACP (optional) to communicate with the new programmable device.

2. Edit SFG$ROOT:[SOURCE.DIM]DEVCOD.MAC, and add the new device manufacturer and model codes.

3. Create a new Device Interface Module (DIM) for the new device (see Appendix C).

4. Edit SFG$ROOT:[SOURCE.DIM]DEVVEC.MAC to include the new DIM.

5. Relink DIRSRV, GENSRV, and POLSRV to incorporate the new DIM.

6. Copy DIRSRV.TSK, GENSRV.TSK, and POLSRV.TSK to SFG$SYSTEM.

7. Modify SFG$SYSTEM:SFGVMR.CMD to load the driver and install the ACP.

8. Invoke SFG$SYSTEM:GATEWAY.COM.

See the SHOP FLOOR GATEWAY Software Installation Guide and Release Notes for help with system tailoring.

D.6 Adding A New Device to BASEWAY

The following steps can be used to add a new device to BASEWAY:

1. Update the module BASE_DEVICE_TYPE_DEFS in BSL$DEFS library to reflect the new manufacturer, device, network, etc.

2. Recompile BSL$ROOT:[SOURCE.LIBRARY]PDAPGMFR.PLI, PDAPGMOD.PLI, PDAPTMFR.PLI, PDAPTMOD.PLI, UARBFMFR.PLI, and UARBFMOD.PLI and insert the resulting object files in BSL$LIBRARY.

3. If the device will support uploading, downloading, and comparing, then create subroutines for each function and add references to these subroutines in BSL$ROOT:[SOURCE.LIBRARY]PDAPDNLOD.PLI, PDAPUPLOD.PLI, and PDAPCOMP.PLI.

If the device will not support these functions, then modify PDAPDNLOD, PDAPUPLOD, and PDAPCOMP to return a PDAP$_NOTSUPPORTED status condition.

4. If the device will support start and stop functions, then modify PDAPSTART and PDAPSTOP to send appropriate requests to the SHOP FLOOR GATEWAY. If the device will not support these functions, then modify PDAPSTART and PDAPSTOP to return a PDAP_$NOTSUPPORTED status condition.

5. If any device-specific errors need to be parsed, modify PDAPERROR to parse it.

6. Add commands to PDAPTRAN to translate ASCII addresses to "internal addresses" that the GATEWAY DIM recognizes.

7. Add code to PDAPNXTAD to find the next valid internal address for this device.

8. Modify BSL$ROOT:[SOURCE.UTILITY.EDTBASLIN]EDTDEVDEF.PLI to reflect new device parameters and network. Relink EDTBASLIN.

APPENDIX E

GLOSSARY

The following terminology is intended to aid the user in understanding various concepts and terminology used in the BASEWAY system.

address - address in a programmable device (device-dependent).

application - set of programs performing a single function. An application may be defined to run on up to 4 VAX processors, but can only be active on a single processor. processors via DECnet.

BASEWAY node - VAX/VMS processor which loads and initializes the GATEWAY and receives data from the SHOP FLOOR GATEWAY.

compiled address - an internal representation of a programmable device.

data identifier - a unique internal identifier that is assigned to a data item at definition time.

data item - represents a unique piece of data associated with a shop floor device. Data items may be gathered automatically from a SHOP FLOOR GATEWAY, or may be updated via a callable subroutine interface. Each data item has a value associated with it.

data name - a unique name assigned by a user to a data item.

device - See Programmable Device. device set - a collection of programmable devices that are connected to a SHOP FLOOR GATEWAY. Up to four device sets may be attached to a single gateway. Although a device set can be defined for up to 4 gateways, it can only be active on a single gateway.

equipment - a polling unit, not necessarily physical equipment.

gateway (data collector, SHOP FLOOR GATEWAY, or SFG) - a PDP-11 processor dedicated to the task of communicating with shop floor devices. Acts as a "gateway" between the shop floor devices and an application. May be up to 4 per BASEWAY.

host node - a VAX/VMS processor which loads, initializes, and receives data from the SHOP FLOOR GATEWAY.

interprocess communication - message-passing facilities which allow two processes to communicate. A message is created and then the data to be sent is stored in it. The message is then sent to the destination by calling the SEND procedure, specifying a destination port.

interprocess message - a message sent from one process to another. These messages have a standard header associated with them.

line - one of the three sets of 8 lines on a single collector. Each set of 8 lines connects to 8 Allen-Bradley Data Highway TM , 8 Modicon Modbus TM networks, or terminal support drivers, for bar-code scanner devices.

message code - a unique message identifier in the range 1--32767.

message port - a value of type PORT represents a system-maintained message queue. PORT values are unique in that they are valid anywhere in the network (including in other applications). Thus, transmission of messages between two processes is completely transparent. Once written, programs can be distributed and redistributed among the network nodes with no changes.

named messsage ports - names can be given to message ports to facilitate communication between jobs. Names exist as systemwide or groupwide logical names equating the port name with a port identifier.

NAU (network addressable unit) - an internal numeric identifier ssociated with a particular data stream within a system. Normally, a ocess has a single NAU allocated to it, but may have more. Up to 127 permanently assigned NAUs and 127 temporary NAUs are available.

NAX - an internal numeric identifier associated with a system. Assigned at system definition time.

network - a series of systems in a DECnet environment.

node - a processor that is connected to other processors via DECnet.

polling set - each machine may have an unlimited number of polling sets, each characterized by a polling frequency (.1 seconds--30 minutes), a starting register address, and the number of consecutive registers to be polled.

process - a single program running on an application. Equivalent to VAX/VMS processes and RSX-11S tasks.

programmable devices - up to 32 or 64 stations on each Modicon Modbus 1 or Allen-Bradley Data Highway TM line, or terminal support driver or bar code scanners.

register - data register or status register. May be bit (or coil), byte, word, longword, or string.

shop floor entity - physical or logical entity in the shop floor that is referenced by a unique name. Each has a unique number assigned to it at definition time. Examples are machines, conveyors, departments, work cells, maintenance cribs, and stations.

system - an application, device set, or gateway. Referenced by a unique name.

INDEX