
EPICS: Input / Output Controller Application Developer's Guide

***EPICS Base Release 3.14.9
7 November 2007***

Martin R. Kraimer, Janet Anderson, Andrew Johnson, Eric Norum
(Argonne National Laboratory)
Jeff Hill (Los Alamos National Laboratory)
Ralph Lange, Benjamin Franksen (BESSY)

Table of Contents

Table of Contents	1
Chapter 1: Introduction	7
1.1. Overview	7
1.2. Acknowledgments	9
Chapter 2: Getting Started	11
2.1. Introduction	11
2.2. Example IOC Application	11
2.3. Channel Access Host Example	13
2.4. iocsh	13
2.5. Building IOC components	14
2.6. makeBaseApp	17
2.7. vxWorks boot parameters	20
2.8. RTEMS boot procedure	21
Chapter 3: EPICS Overview	23
3.1. What is EPICS?	23
3.2. Basic Attributes	23
3.3. Hardware - Software Platforms (Vendor Supplied).....	24
3.4. IOC Software Components	25
3.5. Channel Access	27
3.6. OPI Tools	29
3.7. EPICS Core Software.....	30
Chapter 4: EPICS Build Facility	31
4.1. Overview	31
4.2. Build Requirements	33
4.3. Configuration Definitions	34
4.4. Makefiles	37
4.5. Make.....	39
4.6. Makefile definitions	40
4.7. Table of Makefile definitions.....	62
4.8. Configuration Files	73
4.9. Build Documentation Files	75
4.10.Startup Files.....	76
Chapter 5: Database Locking, Scanning, And Processing	79
5.1. Overview	79
5.2. Record Links	79
5.3. Database Links	80
5.4. Database Locking.....	80
5.5. Database Scanning	81
5.6. Record Processing	82
5.7. Guidelines for Creating Database Links	82
5.8. Guidelines for Synchronous Records.....	84
5.9. Guidelines for Asynchronous Records	85
5.10.Cached Puts.....	87

5.11.putNotify	87
5.12.Channel Access Links	87
Chapter 6: Database Definition	91
6.1. Overview	91
6.2. Brief Summary of Database Definition Syntax	91
6.3. General Rules for Database Definition	92
6.4. Menu	95
6.5. Record Type	95
6.6. Device	100
6.7. Driver	101
6.8. Registrar Declaration	101
6.9. Variable Declaration	102
6.10.Function Declaration	102
6.11.Breakpoint Table	103
6.12.Record Instance	104
6.13.Record Information Item	107
6.14.Record Attribute	107
6.15.Breakpoint Tables - Discussion	107
6.16.Menu and Record Type Include File Generation	109
6.17.dbExpand	112
6.18.dbLoadDatabase	112
6.19.dbLoadRecords	113
6.20.dbLoadTemplate	113
6.21.dbReadTest	115
Chapter 7: IOC Initialization	117
7.1. Overview - Environments requiring a main program	117
7.2. Overview - vxWorks	117
7.3. Overview - RTEMS	118
7.4. iocInit	118
7.5. Changing iocCore fixed limits	120
7.6. TSconfigure	121
7.7. initHooks	121
7.8. Environment Variables	122
7.9. Initialize Logging	123
Chapter 8: Access Security	125
8.1. Overview	125
8.2. Quick Start	125
8.3. User's Guide	126
8.4. Design Summary	131
8.5. Access Security Application Programmer's Interface	133
8.6. Database Access Security	138
8.7. Channel Access Security	140
8.8. Trapping Channel Access Writes	142
8.9. Access Control: Implementation Overview	142
8.10.Structures	144
Chapter 9: IOC Test Facilities	145
9.1. Overview	145
9.2. Database List, Get, Put	145
9.3. Breakpoints	147
9.4. Error Logging	149

9.5. Hardware Reports	149
9.6. Scan Reports	150
9.7. Time Server Report	150
9.8. Access Security Commands	151
9.9. Channel Access Reports	152
9.10. Interrupt Vectors	153
9.11. Miscellaneous	154
9.12. Database System Test Routines	154
9.13. Record Link Reports	155
9.14. Old Database Access Testing	156
9.15. Routines to dump database information	157
Chapter 10: IOC Error Logging	159
10.1. Overview	159
10.2. Error Message Routines	159
10.3. errlog Listeners	161
10.4. errlogThread	162
10.5. console output and message queue size	162
10.6. Status Codes	162
10.7. iocLog	163
Chapter 11: Record Support	165
11.1. Overview	165
11.2. Overview of Record Processing	165
11.3. Record Support and Device Support Entry Tables	166
11.4. Example Record Support Module	167
11.5. Record Support Routines	173
11.6. Global Record Support Routines	177
Chapter 12: Device Support.	181
12.1. Overview	181
12.2. Example Synchronous Device Support Module	181
12.3. Example Asynchronous Device Support Module	183
12.4. Device Support Routines	184
12.5. Extended Device Support	186
Chapter 13: Driver Support.	189
13.1. Overview	189
13.2. Device Drivers	189
Chapter 14: Static Database Access	193
14.1. Overview	193
14.2. Definitions	193
14.3. Allocating and Freeing DBBASE	194
14.4. DBENTRY Routines	195
14.5. Read and Write Database	196
14.6. Manipulating Record Types	197
14.7. Manipulating Field Descriptions	198
14.8. Manipulating Record Attributes	199
14.9. Manipulating Record Instances	200
14.10. Manipulating Menu Fields	202
14.11. Manipulating Link Fields	203
14.12. Manipulating MenuForm Fields	203
14.13. Manipulating Information Items	205

Table of Contents

14.14.Find Breakpoint Table	206
14.15.Dump Routines	206
14.16.Examples	207
Chapter 15: Runtime Database Access.....	209
15.1.Overview	209
15.2.Database Include Files	209
15.3.Runtime Database Access Overview	211
15.4.Database Access Routines	214
15.5.Runtime Link Modification	222
15.6.Channel Access Monitors.....	223
15.7.Lock Set Routines.....	223
15.8.Channel Access Database Links.....	225
Chapter 16: EPICS General Purpose Tasks	229
16.1.Overview	229
16.2.General Purpose Callback Tasks	229
16.3.Task Watchdog.....	232
Chapter 17: Database Scanning	235
17.1.Overview	235
17.2.Scan Related Database Fields.....	235
17.3. Scan Related Software Components	236
17.4.Implementation Overview	239
Chapter 18: IOC Shell	245
18.1.Introduction	245
18.2.IOC Shell Operation	245
18.3.IOC Shell Programming.....	248
Chapter 19: libCom	253
19.1.bucketLib	253
19.2.calc	253
19.3.cppStd	257
19.4.epicsExit	257
19.5.cvtFast	258
19.6.cxxTemplates	258
19.7.dbmf	259
19.8.ellLib	260
19.9.epicsRingBytes	260
19.10.epicsRingPointer.....	261
19.11.epicsTimer	263
19.12. fdmgr	269
19.13.freeList	269
19.14.gpHash	269
19.15.logClient.....	270
19.16.macLib	271
19.17.misc.....	272
Chapter 20: libCom OSI libraries.....	277
20.1.Overview	277
20.2.epicsAssert	278
20.3.epicsEvent.....	278
20.4.epicsFindSymbol	280

20.5.epicsInterrupt.	280
20.6.epicsMath	281
20.7.epicsMessageQueue	281
20.8.epicsMutex	283
20.9.epicsStdlib	285
20.10.epicsStdio	285
20.11.epicsStdioRedirect.	286
20.12.epicsThread	286
20.13.epicsTime	292
20.14.osiPoolStatus	299
20.15.osiProcess	300
20.16.osiSigPipeIgnore	300
20.17.osiSock.h	301
20.18. Device Support Library	301
20.19.vxWorks Specific routines	304
Chapter 21: Registry.	307
21.1.Registry.h	307
21.2.registryRecordType.h	307
21.3.registryDeviceSupport.h	308
21.4.registryDriverSupport.h	308
21.5.registryFunction.h	308
21.6.registerRecordDeviceDriver.c	309
21.7.registerRecordDeviceDriver.pl	309
Chapter 22: Database Structures	311
22.1.Overview	311
22.2.Include Files	311
22.3.Structures.	313
Chapter 23: INDEX	315

Chapter 1: Introduction

1.1 Overview

This document describes the core software that resides in an Input/Output Controller (IOC), one of the major components of EPICS. It is intended for anyone developing EPICS IOC databases and/or new record/device/driver support.

The plan of the book is:

Getting Started

A brief description of how to create EPICS support and ioc applications.

EPICS Overview

An overview of EPICS is presented, showing how the IOC software fits into EPICS.

EPICS Build Facility

This chapter, which was written by Janet Anderson, describes the EPICS build facility including directory structure, environment and system requirements, configuration files, Makefiles, and related build tools.

Database Locking, Scanning, and Processing

Overview of three closely related IOC concepts. These concepts are at the heart of what constitutes an EPICS IOC.

Database Definition

This chapter gives a complete description of the format of the files that describe IOC databases. This is the format used by Database Configuration Tools and is also the format used to load databases into an IOC.

IOC Initialization

A great deal happens at IOC initialization. This chapter removes some of the mystery about initialization.

Access Security

Channel Access Security is implemented in IOCs. This chapter explains how it is configured and also how it is implemented.

IOC Test Facilities

Epics supplied test routines that can be executed via the epics or vxWorks shell.

IOC Error Logging

IOC code can call routines that send messages to a system wide error logger.

Record Support

The concept of record support is discussed. This information is necessary for anyone who wishes to provide customized record and device support.

Device Support

Chapter 1: Introduction

Overview

The concept of device support is discussed. Device support takes care of the hardware specific details of record support, i.e. it is the interface between hardware and a record support module. Device support can directly access hardware or may interface to driver support.

Driver Support

The concepts of driver support is discussed. Drivers, which are not always needed, have no knowledge of records but just take care of interacting with hardware. Guidelines are given about when driver support, instead of just device support, should be provided.

Static Database Access

This is a library that works on both Host and IOC. For IOCs it works and on initialized or uninitialized EPICS databases.

Runtime Database Access

The heart of the IOC software is the memory resident database. This chapter describes the interface to this database.

Device Support Library

A set of routines are provided for device support modules that use shared resources such as VME address space.

EPICS General Purpose Tasks

General purpose callback tasks and task watchdog.

Database Scanning

Database scan tasks, i.e. the tasks that request records to process.

IOC Shell

The EPICS IOC shell is a simple command interpreter which provides a subset of the capabilities of the vxWorks shell.

libCom

EPICS base includes a subdirectory src/libCom, which contains a number of c and c++ libraries that are used by the other components of base. This chapter describes most of these libraries.

libCom OSI

This chapter describes the libraries in libCom that provide Operating System Independent (OSI) interfaces used by the rest of EPICS base. LibCom also contains operating system dependent code that implements the OSI interfaces.

Registry

Under vxWorks osiFindGlobalSymbol can be used to dynamically bind to record, device, and driver support. Since on some systems this always returns failure, a registry facility is provided to implement the binding. The basic idea is that any storage meant to be "globally" accessible must be registered before it can be accessed

Database Structures

A description of the internal database structures.

Other than the overview chapter this document describes only core IOC software. Thus it does not describe other EPICS tools which run in an IOC such as the sequencer. It also does not describe Channel Access.

The reader of this manual should also have the following documents:

- *EPICS Record Reference Manual*, Philip Stanley, Janet Anderson and Marty Kraimer
See LANL Web site for latest version.

- EPICS IOC Software Configuration Management, Marty Kraimer, Andrew Johnson, Janet Anderson, Ralph Lange <http://www.aps.anl.gov/asd/controls/epics/EpicsDocumentation/AppDevManuals/iocScm-3.13.2/index.html>
- *vxWorks Programmer's Guide*, Wind River Systems
- *vxWorks Reference Manual*, Wind River Systems
- RTEMS C User's Guide, Online Applications Research

1.2 Acknowledgments

The basic model of what an IOC should do and how to do it was developed by Bob Dalesio at LANL/GTA. The principle ideas for Channel Access were developed by Jeff Hill at LANL/GTA. Bob and Jeff also were the principle implementers of the original IOC software. This software (called GTACS) was developed over a period of several years with feedback from LANL/GTA users. Without their ideas EPICS would not exist.

During 1990 and 1991, ANL/APS undertook a major revision of the IOC software with the major goal being to provide easily extendible record and device support. Marty Kraimer (ANL/APS) was primarily responsible for designing the data structures needed to support extendible record and device support and for making the changes needed to the IOC resident software. Bob Ziemann (ANL/APS) designed and implemented the UNIX build tools and IOC modules necessary to support the new facilities. Frank Lenkszus (ANL/APS) made extensive changes to the Database Configuration Tool (DCT) necessary to support the new facilities. Janet Anderson developed methods to systematically test various features of the IOC software and is the principal implementer of changes to record support.

During 1993 and 1994, Matt Needes at LANL implemented and supplied the description of fast database links and the database debugging tools.

During 1993 and 1994 Jim Kowalkowski at ANL/APS developed GDCT and also developed the ASCII database instance format now used as the standard format. At that time he also created `dbLoadRecords` and `dbLoadTemplate`.

The `build` utility method resulted in the generation of binary files of UNIX that were loaded into IOCs. As new IOC architectures started being supported this caused problems. During 1995, after learning from an abandoned effort now referred to as `EpicsRX`, the build utilities and binary file (called `default.dctsdcr`) were replaced by all ASCII files. The new method provides architecture independence and a more flexible environment for configuring the record/device/driver support. This principle implementer was Marty Kraimer with many ideas contributed by John Winans and Jeff Hill. Bob Dalesio made sure that we did not go too far, i.e. 1) make it difficult to upgrade existing applications and 2) lose performance.

In early 1996 Bob Dalesio tackled the problem of allowing runtime link modification. This turned into a cooperative development effort between Bob and Marty Kraimer. The effort included new code for database to Channel Access links, a new library for lock sets, and a cleaner interface for accessing database links.

In early 1999 the port of `iocCore` to non `vxWorks` operating systems was started. The principle developers were Marty Kraimer, Jeff Hill, and Janet Anderson. William Lupton converted the sequencer as well as helping with the posix threads implementation of `osiSem` and `osiThread`. Eric Norum provided the port to RTEMS and also contributed the shell that is used on non `vxWorks` environments. Ralph Lange provided the port to HP-UX.

Many other people have been involved with EPICS development, including new record, device, and driver support modules.

Chapter 2: Getting Started

2.1 Introduction

This chapter provides a brief introduction to creating EPICS IOC applications. It contains:

- Instructions for creating, building, and running an example IOC application.
- Instructions for creating, building, and executing example Channel Access clients.
- Briefly describes iocsh, which is a base supplied command shell.
- Describes rules for building IOC components.
- Describes makeBaseApp.pl, which is a perl script that generates files for building applications.
- Briefly discusses vxWorks boot parameters

This chapter will be hard to understand unless you have some familiarity with IOC concepts such as record/device/driver support and have had some experience with creating ioc databases. Once you have this experience, this chapter provides most of the information needed to build applications. The example that follows assumes that EPICS base has already been built.

2.2 Example IOC Application

This section explains how to create an example IOC application in a directory <top>, naming the application myexampleApp and the ioc directory iocmyexample.

2.2.1 Check that EPICS_HOST_ARCH is defined

Execute the command:

```
echo $EPICS_HOST_ARCH          (Unix/linux)
```

or

```
set EPICS_HOST_ARCH            (Windows)
```

This should display your workstation architecture, for example linux-x86 or win32-x86. If you get an "Undefined variable" error, you should set EPICS_HOST_ARCH to your host operating system followed by a dash and then your host architecture, e.g. solaris-sparc. The perl script EpicsHostArch.pl in the base/startup directory has been provided to help set EPICS_HOST_ARCH.

2.2.2 Create the example application

The following commands create an example application.

```
mkdir <top>
```

```
cd <top>
<base>/bin/<arch>/makeBaseApp.pl -t example myexample
<base>/bin/<arch>/makeBaseApp.pl -i -t example myexample
```

Here, <arch> indicates the operating system architecture of your computer. For example, solaris-sparc. The last command will ask you to enter an architecture for the IOC. It provides a list of architectures for which base has been built.

The full path name to <base> (an already built copy of EPICS base) must be given. Check with your EPICS system administrator to see what the path to your <base> is. For example:

```
/home/phoebus/MRK/epics/base/bin/linux-x86/makeBaseApp.pl ...
```

Windows Users Note: Perl scripts are invoked with the command perl <scriptname> on win95/NT. Perl script names are case sensitive. For example to create an application on WIN95/NT:

```
perl C:\epics\base\bin\win32-x86\makeBaseApp.pl -t example myexample
```

2.2.3 Inspect files

Spend some time looking at the files that appear under <top>. Do this BEFORE building. This allows you to see typical files which are needed to build an application without seeing the files generated by make.

2.2.4 Sequencer Example

The sequencer is now supported as an unbundled product. The example includes an example state notation program; sncExample.stt. As created by makeBaseApp the example is not built or executed.

Before sncExample.st can be built, the sequencer must be built using the same version of base that the example uses.

To build sncExample edit the following files:

- configure/RELEASE - Set SNCSEQ to the location of the sequencer.
- iocBoot/iocmyexample/st.cmd - Remove the comment character # from
#seq sncExample,"user=<user>"

The Makefile contains commands for building sncExample as a component of the ioc application and as a standalone application, i.e. an application that does not use an epics database.

2.2.5 Build

In directory <top> execute the command

```
make
```

NOTE: On systems where GNU make is not the default another command is required, e.g. gnumake, gmake, etc. See you EPICS system administrator.

2.2.6 Inspect files

This time you will see the files generated by make as well as the original files.

2.2.7 Run the ioc example

The example can be run on vxWorks, RTEMS, or on a supported host.

- On a host, e.g. linux or solarius
 - `cd <top>/iocBoot/iocmyexample`
 - `../../../../bin/linux-x86/myexample st.cmd`
- vxWorks/RTERMS - Set your boot parameters as described at the end of this chapter and then boot the ioc.

After the ioc is started try some of the shell commands (e.g. `dbl` or `dbpr <recordname>`) described in chapter "IOC Test Facilities". In particular run `dbl` to get a list of the records.

The `iocsh` command interpreter used on non-vxWorks IOCs provides a help facility. Just type:

```
help
or
```

```
help <cmd>
```

where `<cmd>` is one of the commands displayed by `help`. The `help` command accepts wildcards, so

```
help db*
```

will provide information on all commands beginning with the characters `db`.

On vxWorks the help facility is available by first typing:

```
iocsh
```

2.3 Channel Access Host Example

An example host example can be generated by:

```
cd <mytop>
<base>/bin/<arch>/makeBaseApp.pl -t caClient caClient
make
```

(or `gnumake`, as required by your operating system)

Two channel access examples are provided.

- `caExample` - This example accepts a `pvname`, connects and reads the current value for `pvname`, displays the result and terminates. To run this example just type:

```
<mytop>/bin/<hostarch>/caExample <pvname>
```

where
 - `<mytop>` is the full path name to your application top directory.
 - `<hostarch>` is your host architecture.
 - `<pvname>` is one of the record names displayed by the `dbl` ioc shell command.
- `caMonitor` - This example accepts a filename, which contains a list of `pvnames`, each appearing on a separate line. It connects to each `pv` and issues monitor requests. It displays messages for all channel access events, connection events, etc.

2.4 iocsh

Because the vxWorks shell is only available on vxWorks, EPICS base provides `iocsh`. In the main program it can be invoked as follows:

```
iocsh("filename")
```

or

```
iocsh(0)
```

If the argument is a filename, the commands in the file are executed and iocsh returns. If the argument is 0 then iocsh goes into interactive mode, i.e. it prompts for and executes commands until an exit command is issued.

This shell is described in more detail in Chapter 18, "IOC Shell" on page 245

On vxWorks iocsh is not automatically started. It can be started by just giving the following command to the vxWorks shell.

```
iocsh
```

To get back to the vxWorks shell just say

```
exit
```

2.5 Building IOC components

Detailed build rules are given in chapter "Epics Build Facility". This section describes methods for building most components needed for IOC applications. It uses excerpts from the myexampleApp/src/Makefile that is generated by makeBaseApp.

The following two types of applications can be built:

- Support applications
These are applications meant for use by ioc applications. The rules described here install things into one of the following directories that are created just below <top>:
 - include
C include files are installed here. Either header files supplied by the application or header files generated from xxxRecord.dbd or xxxMenu.dbd files.
 - dbd
Each file contains some combination of include, recordtype, device, driver, and registrar database definition commands. The following are installed:
 - xxxRecord.dbd, xxxMenu.dbd files
 - An arbitrary xxx.dbd file
 - ioc applications install a file yyy.dbd generated from file yyyInclude.dbd.
 - db
Files containing record instance definitions.
 - lib/<arch>
All source modules are compiled and placed in shared or static library (win32 dll)
- IOC applications
These are applications loaded into actual IOCs.

2.5.1 Binding to IOC components

Because many IOC components are bound only during ioc initialization, some method of linking to the appropriate shared and/or static libraries must be provided. The method used for IOCs is to generate, from an xxxInclude.dbd file, a C++ program that forces a reference to the appropriate library modules. The following keywords for database definitions are used for this purpose:

```
recordtype  
device  
driver
```

```

    registrar
    variable
    function

```

The method also requires that IOC components call an appropriate `epicsExport` function. Each such component must contain the statement: (NOTE: This must be the last include file in the source module.

```
#include <epicsExport.h>
```

Each record support module must contain a statement like:

```
epicsExportAddress(rset,xxxRSET);
```

Each device support module must contain a statement like:

```
epicsExportAddress(dset,devXxxSoft);
```

Each driver support module must contain a statement like:

```
epicsExportAddress(drvet,drvGpib);
```

The keyword `variable` provides access to integer or double variables, e.g. debugging variables. The `xxxInclude.dbd` file can contain definitions like:

```

    variable(asCaDebug,int)
    variable(myDefaultTimeout,double)

```

The code that defines the variables must include code like:

```

int asCaDebug = 0;
epicsExportAddress(int,asCaDebug);

```

The keyword `registrar` signifies that the epics component supplies a C function that has the prototype:

```
typedef void (*REGISTRAR)(void);
```

This function normally registers things described in chapter "Registry". The `myexampleApp` provides an example set of functions for a subroutine record. It contains the statements:

```

static registryFunctionRef mySubRef[] = {
    {"mySubInit", (REGISTRYFUNCTION)mySubInit},
    {"mySubProcess", (REGISTRYFUNCTION)mySubProcess}
};

static void mySub(void)
{
    registryFunctionRefAdd(mySubRef, NELEMENTS(mySubRef));
}
epicsExportRegistrar(mySub);

```

`epicsExportRegistrar` can also be used to register iocsh commands, etc.

The registrar function in the preceding example does nothing but register functions. Functions can also be registered by using the `epicsRegisterFunction` macro in the C source file containing the function and the `function` statement in the application database description file. The example could thus be simplified to:

```

epicsRegisterFunction(mySubInit);
epicsRegisterFunction(mySubProcess);

```

NOTE: The `epicsExport` system requires that the `epicsExportAddress`, `epicsExportRegistrar` or `epicsRegisterFunction` statement appear in the same source module as whatever it exports.

2.5.2 Makefile rules

2.5.2.1 Building a support application.

```
# xxxRecord.h will be created from xxxRecord.dbd
DBDINC += xxxRecord
DBD += myexampleSupport.dbd

LIBRARY_IOC += myexampleSupport

myexampleSupport_SRCS += xxxRecord.c
myexampleSupport_SRCS += devXxxSoft.c
myexampleSupport_SRCS += dbSubExample.c
```

```
myexampleSupport_LIBS += $(EPICS_BASE_IOC_LIBS)
```

The DBDINC rule looks for a file xxxRecord.dbd. From this file a file xxxRecord.h is created and installed into <top>/include

The DBD rule finds myexampleSupport.dbd in the source directory and installs it into <top>/dbd

The LIBRARY_IOC statement states that a shared/static library should be created and installed into <top>/lib/<arch>.

The myexampleSupport_SRCS statements name all the source files that are compiled and put into the library.

The above statements are all that is needed for building many support applications.

2.5.2.2 Building the IOC application

The following statements build the IOC application:

```
PROD_IOC = myexample
```

```
DBD += myexample.dbd
```

```
# myexample.dbd will be made up from these files:
myexample_DBD += base.dbd
myexample_DBD += xxxSupport.dbd
myexample_DBD += dbSubExample.dbd
```

```
# <name>_registerRecordDeviceDriver.cpp will be created from <name>.dbd
myexample_SRCS += myexample_registerRecordDeviceDriver.cpp
myexample_SRCS_DEFAULT += myexampleMain.cpp
myexample_SRCS_vxWorks += -nil-
```

```
# Add locally compiled object code
myexample_SRCS += dbSubExample.c
```

```
#The following adds support from base/src/vxWorks
myexample_OBJS_vxWorks += $(EPICS_BASE_BIN)/vxComLibrary
```

```
myexample_LIBS += myexampleSupport
myexample_LIBS += $(EPICS_BASE_IOC_LIBS)
```

PROD_IOC gives that name of the ioc application, which is named myexample.

The DBD definition `myexample.dbd` will cause build rules to create the database definition include file `myexampleInclude.dbd` from files in the `myexample_DBD` definition. For each filename in the `myexample_DBD` definition, the created `myexampleInclude.dbd` will contain an include statement for that filename. The created `myexampleInclude.dbd` file will contain the following lines.

```
include "base.dbd"
include "xxxSupport.dbd"
include "dbSubExample.dbd"
```

When the DBD build rules find the created file `myexampleInclude.dbd`, the rules then call `dbExpand` which reads `myexampleInclude.dbd` to generate file `myexample.dbd`, and install it into `<top>/dbd`.

An arbitrary number of `myexample_SRCS` statements can be given. One, `myexample_registerRecordDeviceDriver.cpp`, is special. When this is seen the following happens:

- A perl script `registerRecordDeviceDriver.pl` is executed. Taking `myexample.dbd` as input it generates `myexample_registerRecordDeviceDriver.cpp`.

2.6 *makeBaseApp*

`makeBaseApp` is a perl script that creates application areas. It can create the following:

- `<top>/Makefile`
- `<top>/configure` - This directory contains the files needed by the EPICS build system.
- `<top>/xxxApp` - A set of directories and associated files for a major sub-module.
- `<top>/iocBoot` - A subdirectory and associated files.
- `<top>/iocBoot/iocxxx` - A subdirectory and files for a single ioc.

`makeBaseApp` creates directories and then copies template files into the newly created directories while expanding macros in the template files. EPICS base provides two sets of template files: simple and example. These are meant for simple applications. Each site, however, can create its own set of template files which may provide additional functionality. This section describes the functionality of `makeBaseApp` itself, the next section provides details about the simple and example templates.

2.6.1 Usage

`makeBaseApp` has four possible forms of command line:

```
<base>/bin/<arch>/makeBaseApp.pl -h
```

Provides help.

```
<base>/bin/<arch>/makeBaseApp.pl -l [options]
```

List the application templates available. This invocation does not alter the current directory.

```
<base>/bin/<arch>/makeBaseApp.pl [-t type] [options] app ...
```

Create application directories.

```
<base>/bin/<arch>/makeBaseApp.pl -i -t type [options] ioc ...
```

Create ioc boot directories.

Options for all command forms:

```
-b base
```

Chapter 2: Getting Started

makeBaseApp

Provides the full path to EPICS base. If not specified, the value is taken from the EPICS_BASE entry in config/RELEASE. If the config directory does not exist, the path is taken from the command-line that was used to invoke makeBaseApp

-T template

Set the template top directory (where the application templates are). If not specified, the template path is taken from the TEMPLATE_TOP entry in config/RELEASE. If the config directory does not exist the path is taken from the environment variable EPICS_MBA_TEMPLATE_TOP, or if this is not set the templates from EPICS base are used.

-d

Verbose output (useful for debugging)

Arguments unique to makeBaseApp.pl [-t type] [options] app ...:

app

One or more application names (the created directories will have " App" appended to this name)

-t type

Set the template type (use the -l invocation to get a list of valid types). If this option is not used, type is taken from the environment variable EPICS_MBA_DEF_APP_TYPE, or if that is not set the values "default" and then "example " are tried.

Arguments unique to makeBaseApp.pl -i [options] ioc ...:

ioc

One or more IOC names (the created directories will have "ioc " prepended to this name).

-a arch

Set the IOC architecture (e.g. vxWorks-68040). If -a arch is not specified, you will be prompted.

2.6.2 Environment Variables:

EPICS_MBA_DEF_APP_TYPE

Application type you want to use as default

EPICS_MBA_TEMPLATE_TOP

Template top directory

2.6.3 Description

To create a new <top> issue the commands:

```
mkdir <top>
```

```
cd <top>
```

```
<base>/bin/<arch>/makeBaseApp.pl -t <type> <app> ...
```

```
<base>/bin/<arch>/makeBaseApp.pl -i -t <type> <ioc> ...
```

makeBaseApp does the following:

- EPICS_BASE is located by checking the following in order:
 - If the -b option is specified it is used.
 - If a <top>/config/RELEASE file exists and defines a value for EPICS_BASE it is used.
 - It is obtained from the invocation of makeBaseApp. For this to work, the full path name to the makeBaseApp.pl script in the EPICS base release you are using must be given.
 - TEMPLATE_TOP is located in a similar fashion:
 - If the -T option is specified it is used.
 - If a <top>/config/RELEASE file exists and defines a value for TEMPLATE_TOP it is used.
-

- If `EPICS_MBA_TEMPLATE_TOP` is defined it is used.
- It is set equal to `<epics_base>/templates/makeBaseApp/top`
- If `-l` is specified the list of application types is listed and `makeBaseApp` terminates.
- If `-i` is specified and `-a` is not then the user is prompted for the IOC architecture.
- The application type is determined by checking the following in order:
 - If `-t` is specified it is used.
 - If `EPICS_MBA_DEF_APP_TYPE` is defined it is used.
 - If a template `defaultApp` exists, the application type is set equal to `default`.
 - If a template `exampleApp` exists, the application type is set equal to `example`.
- If the application type is not found in `TEMPLATE_TOP`, `makeBaseApp` issues an error and terminates.
- If `Makefile` does not exist, it is created.
- If directory `configure` does not exist, it is created and populated with all the `configure` files.
- If `-i` is specified:
 - If directory `iocBoot` does not exist, it is created and the files from the template boot directory are copied into it.
 - For each `<ioc>` specified on the command line a directory `iocBoot/ioc<ioc>` is created and populated with the files from the template (with `ReplaceLine()` tag replacement, see below).
- If `-i` is NOT specified:
 - For each `<app>` specified on the command line a directory `<app>App` is created and populated with the directory tree from the template (with `ReplaceLine()` tag replacement, see below).

2.6.4 Tag Replacement within a Template

When copying certain files from the template to the new application structure, `makeBaseApp` replaces some predefined tags in the name or text of the files concerned with values that are known at the time. An application template can extend this functionality as follows:

- Two perl subroutines are defined within `makeBaseApp`:
 - `ReplaceFilename` - This substitutes for the following in names of any file taken from the templates.
 - `_APPNAME_`
 - `_APPTYPE_`
 - `ReplaceLine` - This substitutes for the following in each line of each file taken from the templates:
 - `_USER_`
 - `_EPICS_BASE_`
 - `_ARCH_`
 - `_APPNAME_`
 - `_APPTYPE_`
 - `_TEMPLATE_TOP_`
 - `_IOC_`
- If the application type directory has a file named `Replace.pl`, it can:
 - Replace one or both of the above subroutines with its own versions.
 - Add a subroutine `ReplaceFilenameHook($file)` which is called at the end of `ReplaceFilename`.
 - Add a subroutine `ReplaceLineHook($line)` which is called at the end of `ReplaceLine`.
 - Include other code which is run after the command line options are interpreted.

2.6.5 makeBaseApp templates provided with base

2.6.5.1 support

This creates files appropriate for building a support application.

2.6.5.2 ioc

Without the `-i` option, this creates files appropriate for building an ioc application, With the `-i` option it creates an ioc boot directory.

2.6.5.3 example

Without the `-i` option it creates files for running an example. Both a support and an ioc application are built. With the `-i` option it creates an ioc boot directory that can be used to run the example.

2.6.5.4 caClient

This builds two Channel Access clients.

2.6.5.5 caServer

This builds an example Portable Access Server.

2.7 vxWorks boot parameters

The vxWorks boot parameters are set via the console serial port on your IOC. Life is much easier if you find out how to connect the serial port to a window on your workstation.

The vxWorks boot parameters look something like the following:

```
boot device           : xxx
processor number      : 0
host name             : xxx
file name            : <full path to board support>/vxWorks
inet on ethernet (e) : xxx.xxx.xxx.xxx:<netmask>
host inet (h)        : xxx.xxx.xxx.xxx
user (u)             : xxx
ftp password (pw)    : xxx
flags (f)            : 0x0
target name (tn)     : <hostname for this inet address>
startup script (s)   : <top>/iocBoot/iocmyexample/st.cmd
```

The actual values for each field are site and IOC dependent. Two fields that you can change at will are the vxWorks boot image and the location of the startup script.

Note that the full path name for the correct board support boot image must be specified. If bootp is used the same information will need to be placed in the bootp host's configuration database instead.

When your boot parameters are set properly, just press the reset button on your IOC, or use the `@` command to commence booting. You will find it VERY convenient to have the console port of the IOC attached to a scrolling window on your workstation.

2.8 RTEMS boot procedure

RTEMS uses the vendor-supplied bootstrap mechanism so the method for booting an IOC depends upon the hardware in use.

2.8.1 Booting from a BOOTP/DHCP/TFTP server

Many boards can use BOOTP/DHCP to read their network configuration and then use TFTP to read the application program. RTEMS can then use TFTP or NFS to read startup scripts and configuration files. If you are using TFTP to read the startup scripts and configuration files you must install the EPICS application files on your TFTP server as follows:

- Copy all db/xxx files to <tftpbase>/epics/<target_hostname>/db/xxx.
- Copy all dbd/xxx files to <tftpbase>/epics/<target_hostname>/dbd/xxx.
- Copy the st.cmd script to <tftpbase>/epics/<target_hostname>/st.cmd.

Use DHCP site-specific option 129 to specify the path to the IOC startup script.

2.8.2 Motorola PPCBUG boot parameters

Motorola single-board computers which employ PPCBUG should have their 'NIOT' parameters set up like:

```

Controller LUN =00
Device LUN      =00
Node Control Memory Address =FFE10000
Client IP Address      ='Dotted-decimal' IP address of IOC
Server IP Address      ='Dotted-decimal' IP address of TFTP/NFS server
Subnet IP Address Mask ='Dotted-decimal' IP address of subnet mask (255.255.255.0 for class C subnet)
Broadcast IP Address   ='Dotted-decimal' IP address of subnet broadcast address
Gateway IP Address     ='Dotted-decimal' IP address of network gateway (0.0.0.0 if none)
Boot File Name         =Path to application bootable image (.../bin/RTEMS-mvme2100/test.boot)
Argument File Name     =Path to application startup script (.../iocBoot/ioctest/st.cmd)
Boot File Load Address =001F0000 (actual value depends on BSP)
Boot File Execution Address =001F0000 (actual value depends on BSP)
Boot File Execution Delay =00000000
Boot File Length       =00000000
Boot File Byte Offset  =00000000
BOOTP/RARP Request Retry =00
TFTP/ARP Request Retry  =00
Trace Character Buffer Address =00000000

```

2.8.3 Motorola MOTLOAD boot parameters

Motorola single-board computers which employ MOTLOAD should have their network 'Global Environment Variable' parameters set up like:

```

mot- /dev/enet0-cipa='Dotted-decimal' IP address of IOC
mot- /dev/enet0-sipa='Dotted-decimal' IP address of TFTP/NFS server
mot- /dev/enet0-snma='Dotted-decimal' IP address of subnet mask (255.255.255.0 for class C subnet)
mot- /dev/enet0-gipa='Dotted-decimal' IP address of network gateway (omit if none)
mot- /dev/enet0-file=Path to application bootable image (.../bin/RTEMS-mvme5500/test.boot)
rtems-client-name=IOC name (mot- /dev/enet0-cipa will be used if this parameter is missing)
rtems-dns-server='Dotted-decimal' IP address of domain name server (omit if none)

```

`rtems-dns-domainname=Domain name (if this parameter is omitted the compiled-in value will be used)`
`epics-script=Path to application startup script (..../iocBoot/iocTest/st.cmd)`

The `mot-script-boot` parameter should be set up like:

```
tftpGet -a4000000 -cxxx -sxxx -mxxx -gxxx -d/dev/enet0 -f....bin/RTEMS-mvme5500/test.boot
netShut
go -a4000000
```

where the `-c`, `-s`, `-m` and `-g` values should match the `cipa`, `sipa`, `snma` and `gipa` values, respectively and the `-f` value should match the file value.

2.8.4 RTEMS NFS access

For IOCs which use NFS for remote file access the EPICS initialization code uses the startup script pathname to determine the parameters for the initial NFS mount. If the startup script pathname begins with a `'/'` the first component of the pathname is used as both the server path and the local mount point. If the startup script pathname does not begin with a `'/'` the first component of the pathname is used as the local mount point and the server path is `"/tftpboot/"` followed by the first component of the pathname. This allows the NFS client used for EPICS file access and the TFTP client used for bootstrapping the application to have a similar view of the remote filesystem.

2.8.5 RTEMS 'Cexp'

The RTEMS 'Cexp' add-on package provides the ability to load object modules at application run-time. If your RTEMS build includes this package you can load RTEMS IOC applications in the same fashion as vxWorks IOC applications.

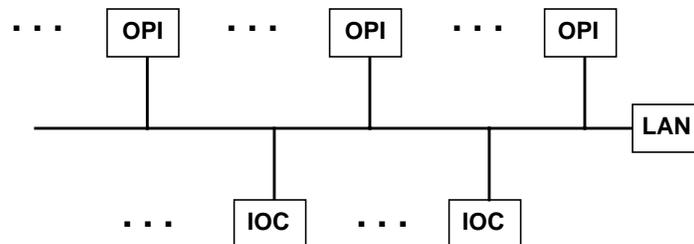
Chapter 3: EPICS Overview

3.1 What is EPICS?

EPICS consists of a set of software components and tools that Application Developers use to create a control system. The basic components are:

- **OPI:** Operator Interface. This is a workstation which can run various EPICS tools.
- **IOC:** Input/Output Controller. Any platform that can support EPICS run time databases together with the other software components described in the manual. One example is a workstation. Another example is a VME/VXI based system using vxWorks or RTEMS as the realtime operating system.
- **LAN:** Local Area Network. This is the communication network which allows the IOCs and OPIs to communicate. EPICS provides a software component, Channel Access, which provides network transparent communication between a Channel Access client and an arbitrary number of Channel Access servers.

A control system implemented via EPICS has the following physical structure.



The rest of this chapter gives a brief description of EPICS:

- **Basic Attributes:** A few basic attributes of EPICS.
- **Platforms:** The vendor supplied Hardware and Software platforms EPICS supports.
- **IOC Software:** EPICS supplied IOC software components.
- **Channel Access:** EPICS software that supports network independent access to IOC databases.
- **OPI Tools:** EPICS supplied OPI based tools.
- **EPICS Core:** A list of the EPICS core software, i.e. the software components without which EPICS will not work.

3.2 Basic Attributes

The basic attributes of EPICS are:

- **Tool Based:** EPICS provides a number of tools for creating a control system. This minimizes the need for custom coding and helps ensure uniform operator interfaces.
- **Distributed:** An arbitrary number of IOCs and OPIs can be supported. As long as the network is not saturated, no single bottle neck is present. A distributed system scales nicely. If a single IOC becomes saturated, its functions can be spread over several IOCs. Rather than running all applications on a single host, the applications can be spread over many OPIs.
- **Event Driven:** The EPICS software components are all designed to be event driven to the maximum extent possible. For example, rather than having to poll IOCs for changes, a Channel Access client can request that it be notified when a change occurs. This design leads to efficient use of resources, as well as, quick response times.
- **High Performance:** A SPARC based workstation can handle several thousand screen updates a second with each update resulting from a Channel Access event. A 68040 IOC can process more than 6,000 records per second, including generation of Channel Access events.

3.3 Hardware - Software Platforms (Vendor Supplied)

EPICS core components (including IOC components) run on a wide range of systems. Currently this includes the following platforms, but new operating system platforms can be easily supported if they have reasonable support for sockets and threads. Currently most 32 bit processors are supported. Some limited testing has been performed on 64 bit processors.

3.3.1 OPI

Platforms

- Unix based Workstations: Well supported platforms include SOLARIS, and HP-UX
- Linux
- Darwin, i.e. Mac OS 10
- Windows NT
- Limited support for VMS

3.3.2 LAN

Hardware

- Ethernet (most flavors)

Software

- TCP/IP protocols via sockets

3.3.3 IOC

Hardware

- VME/VXI bus and crates
 - Various VME modules (ADCs, DAC, Binary I/O, etc.)
 - Allen Bradley Scanner (Most AB I/O modules)
 - GPIB devices

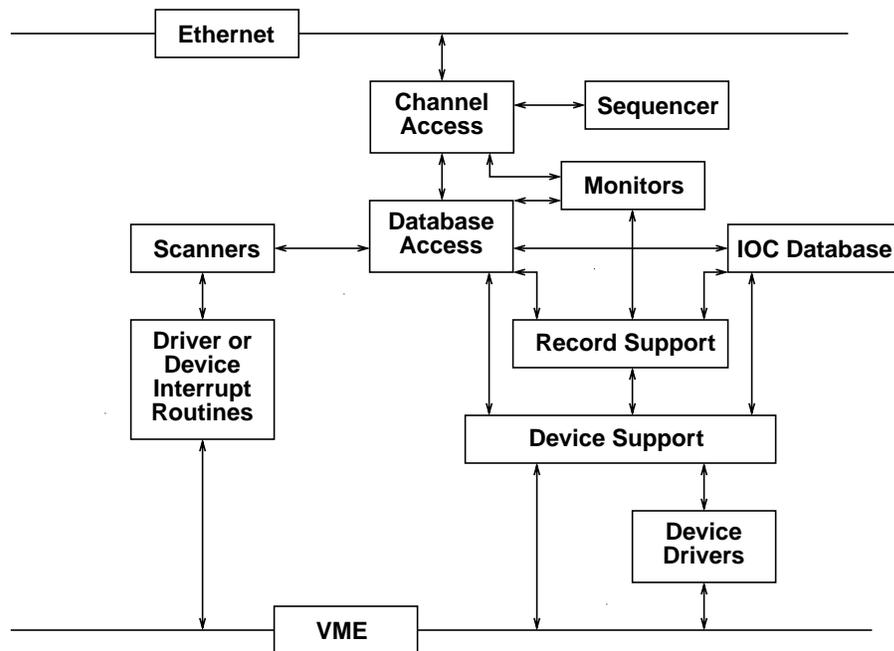
- BITBUS devices
- CAMAC
- CANBUS
- Motorola 68K
- Intel
- AMD Athelon
- PowerPC
- Sun Sparc
- HP

Software

- vxWorks operating system
 - Real time kernel
 - Extensive “Unix like” libraries
- RTEMS
- Linux
- Unix
- Darwin
- win32

3.4 IOC Software Components

An IOC contains the following EPICS supplied software components.



- **IOC Database:** The memory resident database plus associated data structures.

- **Database Access:** Database access routines. With the exception of record and device support, all access to the database is via the database access routines.
- **Scanners:** The mechanism for deciding when records should be processed.
- **Record Support:** Each record type has an associated set of record support routines.
- **Device Support:** Each record type can have one or more sets of device support routines.
- **Device Drivers:** Device drivers access external devices. A driver may have an associated driver interrupt routine.
- **Channel Access:** The interface between the external world and the IOC. It provides a network independent interface to database access.
- **Monitors:** Database monitors are invoked when database field values change.
- **Sequencer:** A finite state machine.

Let's briefly describe the major components of the IOC and how they interact.

3.4.1 IOC Database

The heart of each IOC is a memory resident database together with various memory resident structures describing the contents of the database. EPICS supports a large and extensible set of record types, e.g. `ai` (Analog Input), `ao` (Analog Output), etc.

Each record type has a fixed set of fields. Some fields are common to all record types and others are specific to particular record types. Every record has a record name and every field has a field name. The first field of every database record holds the record name, which must be unique across all IOCs that are attached to the same TCP/IP subnet.

Data structures are provided so that the database can be accessed efficiently. Most software components, because they access the database via database access routines, do not need to be aware of these structures.

3.4.2 Database Access

With the exception of record and device support, all access to the database is via the channel or database access routines. See Chapter 15, "Runtime Database Access" on page 209 for details.

3.4.3 Database Scanning

Database scanning is the mechanism for deciding when to process a record. Five types of scanning are possible: Periodic, Event, I/O Event, Passive and Scan Once.

- **Periodic:** A request can be made to process a record periodically. A number of time intervals are supported.
- **Event:** Event scanning is based on the posting of an event by any IOC software component. The actual subroutine call is:

```
post_event(event_num)
```
- **I/O Event:** The I/O event scanning system processes records based on external interrupts. An IOC device driver interrupt routine must be available to accept the external interrupts.
- **Passive:** Passive records are processed as a result of linked records being processed or as a result of external changes such as Channel Access puts.
- **Scan Once:** In order to provide for caching puts, The scanning system provides a routine `scanOnce` which arranges for a record to be processed one time.

3.4.4 Record Support, Device Support and Device Drivers

Database access needs no record-type specific knowledge, because each record-type has its associated record support module. Therefore, database access can support any number and type of records. Similarly, record support contains no device specific knowledge, giving each record type the ability to have any number of independent device support modules. If the method of accessing the piece of hardware is more complicated than what can be handled by device support, then a device driver can be developed.

Record types *not* associated with hardware do not have device support or device drivers.

The IOC software is designed so that the database access layer knows nothing about the record support layer other than how to call it. The record support layer in turn knows nothing about its device support layer other than how to call it. Similarly the only thing a device support layer knows about its associated driver is how to call it. This design allows a particular installation and even a particular IOC within an installation to choose a unique set of record types, device types, and drivers. The remainder of the IOC system software is unaffected.

Because an Application Developer can develop record support, device support, and device drivers, these topics are discussed in greater detail in later chapters.

Every record support module must provide a record processing routine to be called by the database scanners. Record processing consists of some combination of the following functions (particular records types may not need all functions):

- **Input:** Read inputs. Inputs can be obtained, via device support routines, from hardware, from other database records via database links, or from other IOCs via Channel Access links.
- **Conversion:** Conversion of raw input to engineering units or engineering units to raw output values.
- **Output:** Write outputs. Output can be directed, via device support routines, to hardware, to other database records via database links, or to other IOCs via Channel Access links.
- **Raise Alarms:** Check for and raise alarms.
- **Monitor:** Trigger monitors related to Channel Access callbacks.
- **Link:** Trigger processing of linked records.

3.4.5 Channel Access

Channel Access is discussed in the next section.

3.4.6 Database Monitors

Database monitors provide a callback mechanism for database value changes. This allows the caller to be notified when database values change without constantly polling the database. A mask can be set to specify value changes, alarm changes, and/or archival changes.

At the present time only Channel Access uses database monitors. No other software should use the database monitors. The monitor routines will not be described because they are of interest only to Channel Access.

3.5 Channel Access

Channel Access provides network transparent access to IOC databases. It is based on a client/ server model. Each IOC provides a Channel Access server which is willing to establish communication with an arbitrary number of clients. Channel Access client services are available on both OPIs and IOCs. A client can communicate with an arbitrary number of servers.

3.5.1 Client Services

The basic Channel Access client services are:

- **Search:** Locate the IOCs containing selected process variables and establish communication with each one.
- **Get:** Get value plus additional optional information for a selected set of process variables.
- **Put:** Change the values of selected process variables.
- **Add Event:** Add a change of state callback. This is a request to have the server send information only when the associated process variable changes state. Any combination of the following state changes can be requested: change of value, change of alarm status and/or severity, and change of archival value. Many record types provide hysteresis factors for value changes.

In addition to requesting process variable values, any combination of the following additional information may be requested:

- **Status:** Alarm status and severity.
- **Units:** Engineering units for this process variable.
- **Precision:** Precision with which to display floating point numbers.
- **Time:** Time when the record was last processed.
- **Enumerated:** A set of ASCII strings defining the meaning of enumerated values.
- **Graphics:** High and low limits for producing graphs.
- **Control:** High and low control limits.
- **Alarm:** The alarm HIHI, HIGH, LOW, and LOLO values for the process variable.

It should be noted that Channel Access does *not* provide access to database records as records. This is a deliberate design decision. This allows new record types to be added without impacting any software that accesses the database via Channel Access, and it allows a Channel Access client to communicate with multiple IOCs having differing sets of record types.

3.5.2 Search Server

Channel Access provides an IOC resident server which waits for Channel Access search messages. These are generated when a Channel Access client (for example when an Operator Interface task starts) searches for the IOCs containing process variables the client uses. This server accepts all search messages, checks to see if any of the process variables are located in this IOC, and, if any are found, replies to the sender with an “I have it” message.

3.5.3 Connection Request Server

Once the process variables have been located, the Channel Access client issues connection requests for each IOC containing process variables the client uses. The connection request server, in the IOC, accepts the request and establishes a connection to the client. Each connection is managed by two separate tasks: `ca_get` and `ca_put`. The `ca_get` and `ca_put` requests map to `dbGetField` and `dbPutField` database access requests. `ca_add_event` requests result in database monitors being established. Database access and/or record support routines trigger the monitors via a call to `db_post_event`.

3.5.4 Connection Management

Each IOC provides a connection management service. When a Channel Access server fails (e.g. its IOC crashes) the client is notified and when a client fails (e.g. its task crashes) the server is notified. When a client fails, the server breaks the connection. When a server crashes, the client automatically re-establishes communication when the server restarts.

3.6 OPI Tools

EPICS provides a number of OPI based tools. These can be divided into two groups based on whether or not they use Channel Access. Channel Access tools are real time tools, i.e. they are used to monitor and control IOCs.

3.6.1 Examples of channel Access Tools

A large number of Channel Access tools have been developed. The following are some representative examples.

- **EDM** - Extensible Display Manager. The newest display manager/editor for EPICS.
- **MEDM**: Motif version of combined display manager and display editor.
- **DM**: Display Manager. Reads one or more display list files created by EDD, establishes communication with all necessary IOCs, establishes monitors on process variables, accepts operator control requests, and updates the display to reflect all changes.
- **stripTool** - General purpose stripchart tool.
- **ALH**: Alarm Handler. General purpose alarm handler driven by an alarm configuration file.
- **AR**: Archiver. General purpose tool to acquire and save data from IOCs.
- **Sequencer**: Runs in an IOC and emulates a finite state machine.
- **BURT**: Backup and Restore Tool. General purpose tool to save and restore Channel Access channels. The tool can be run via Unix commands or via a Graphical User Interface.
- **KM**: Knob Manager - Channel Access interface for the sun dials (a set of 8 knobs)
- **PROBE**: Allows the user to monitor and/or change a single process variable specified at run time.
- **CAMATH**: Channel Access interface for Mathematica.
- **CAWINGZ**: Channel Access interface for Wingz.
- **IDL/PVWAVE** Channel Access Interfaces exist for these products.
- **TCL/TK** Channel Access Interface for these products.
- **CDEV** - A library designed to provide a standard API to one or more underlying packages, typically control system interfaces. CDEV provides a Channel Access service.

3.6.2 Examples of other OPI Tools

- **VDCT** - A Java based database configuration tool which is quickly becoming the recommended database configuration tool.
- **JDCT**: Java Database Configuration Tool. A JAVA based tool for creating run time databases.
- **GDCT**: Graphical Database Configuration Tool. Used to create a run time database for an IOC. This is no longer being developed since it is based on an open source software system called unidraw, which is no longer being supported.
- **EDD**: Display Editor. This tool is used to create a display list file for the Display Manager. A display list file contains a list of static, monitor, and control elements. Each monitor and control element has an associated process variable.
- **SNC**: State Notation Compiler. It generates a C program that represents the states for the IOC Sequencer tool.
- **Database Tools** - Tools are provided which generate C include files from menu and record type database definition files.
- **Source/Release**: EPICS provides a Source/Release mechanism for managing EPICS.

3.7 EPICS Core Software

EPICS consists of a set of core software and a set of optional components. The core software, i.e. the components of EPICS without which EPICS would not function, are:

- Channel Access - Client and Server software
- IOC Database
- Scanners
- Monitors
- Database Definition Tools
- Source/Release

All other software components are optional. Of course, any application developer would be crazy to ignore tools such as MEDM (or EDD/DM). Likewise an application developer would not start from scratch developing record and device support. Most OPI tools do not, however, have to be used. Likewise any given record support module, device support module, or driver could be deleted from a particular IOC and EPICS will still function.

Chapter 4: EPICS Build Facility

Janet Anderson is the author of this chapter.

4.1 Overview

This chapter describes the EPICS build facility including directory structure, environment and system requirements, configuration files, Makefiles, and related build tools.

4.1.1 <top> Directory structure

EPICS software can be divided into multiple <top> areas. Examples of <top> areas are EPICS base itself, EPICS extensions, and simple or complicated IOC applications. Each <top> may be maintained separately. Different <top> areas can be on different releases of external software such as EPICS base releases.

A <top> directory has the following directory structure:

```
<top>/
  Makefile
  configure/
  dir1/
  dir2/
  ...
```

where configure is a directory containing build configuration files and a Makefile and where dir1, dir2, ... are user created subdirectory trees with Makefiles and source files to be built.

4.1.2 Install Directories

Files installed during the build are installed into subdirectories of an installation directory which defaults to \$(TOP), the <top> directory. For base the default value can be changed in the configure/CONFIG_SITE file. For extensions and applications the default value can be overridden in the configure/CONFIG file. The installation directories for the EPICS components are:

- **INSTALL_LOCATION** - base.
- **INSTALL_LOCATION_EXTENSIONS** - extensions
- **INSTALL_LOCATION_APP** - applications

The following subdirectories may also exist in the installation directory. They are created by the build and contain the installed build components.

- **dbd** - Directory into which Database Definition files are installed.
- **include** - The directory into which C header files are installed. These header files may be generated from menu and record type definitions.

- **bin** - This directory contains a subdirectory for each host architecture and for each target architecture. These are the directories into which executables, binaries, etc. are installed.
- **lib** - This directory contains a subdirectory for each host architecture. These are the directories into which libraries are installed.
- **db** - This is the directory into which database record instance, template, and substitution files are installed.
- **html** - This is the directory into which html documentation is installed.
- **templates** - This is the directory into which template files are installed.
- **javalib** - This is the directory into which java class files and jar files are installed.
- **configure** - The directory into which configure files are installed (if `INSTALL_LOCATION` not equal `TOP`).

4.1.3 Elements of build system

The main ingredients of the build system are:

- A set of configuration files and tools provided in the EPICS base/configure directory
- A corresponding set of configuration files in the `<top>/configure` directory of a non-base `<top>` directory structure to be built. The `makeBaseApp.pl` and `makeBaseExt.pl` scripts create these configuration files. Many of these files just include a file of the same name from the base/configure directory.
- Makefiles in each directory of the `<top>` directory structure to be built

4.1.4 Features

The principal features of the build system are:

- Requires a single Makefile in each directory of a `<top>` directory structure
- Supports both host os vendor's native compiler and GNU compiler
- Supports building multiple types of software (libraries, executables, databases, java class files, etc.) stored in a single directory tree.
- Supports building EPICS base, extensions, and IOC applications.
- Supports multiple host and target operating system - architecture combinations.
- Allows builds for all hosts and targets within a single `<top>` source directory tree.
- Allows sharing of components such as special record/device/drivers across `<top>` areas.
- `gnumake` is the only command used to build a `<top>` area.

4.1.5 Multiple host and target systems

You can build on multiple host systems and for multiple cross target systems using a single EPICS directory structure. The intermediate and binary files generated by the build will be created in separate `O.*` subdirectories and installed into the appropriate separate host or target install directories. EPICS executables and scripts are installed into the `$(INSTALL_LOCATION)/bin/<arch>` directories. Libraries are installed into `$(INSTALL_LOCATION)/lib/<arch>`. The default definition for `$(INSTALL_LOCATION)` is `$(TOP)` which is the root directory in the directory structure. Architecture dependant created files (e.g. object files) are stored in `O.<arch>` source subdirectories, and architecture independent created files are stored in `O.Common` source subdirectories. This allows objects for multiple cross target architectures to be maintained at the same time.

To build EPICS base for a specific host/target combination you must have the proper host/target `c/c++` cross compiler and target header files, `CROSS_COMPILER_HOST_ARCHS` must empty or include the host architecture in its list value, the `CROSS_COMPILER_TARGET_ARCHS` variable must include the target to be cross-compiled, and the `base/configure/os` directory must have the appropriate configure files.

4.2 Build Requirements

4.2.1 Host Environment Variable

Only one environment variable, `EPICS_HOST_ARCH`, is required to build EPICS <top> areas. This variable should be set to be your workstation's operating system - architecture combination to use the os vendor's c/c++ compiler for native builds or set to the operating system - architecture - alternate compiler combination to use an alternate compiler for native builds if an alternate compiler is supported on your system. The filenames of the `CONFIG.*.Common` files in `base/configure/os` show the currently supported `EPICS_HOST_ARCH` values. Examples are `solaris-sparc`, `solaris-sparc-gnu`, `linux-x86`, `win32-x86`, and `win32-x86-borland`.

4.2.2 Software Prerequisites

Before you can build EPICS components your host system must have the following software installed:

- Perl version 5 or greater
- GNU make, version 3.78.1 or greater
- C++ compiler (host operating system vendor's compiler or GNU compiler)

If you will be building EPICS components for vxWorks targets you will also need:

- Tornado II and one or more board support packages. Consult the vxWorks documentation for details.

If you will be building EPICS components for RTEMS targets you will also need:

- RTEMS development tools and libraries required to run EPICS IOC applications.

4.2.3 Path requirements

You must have the perl executable in your path and you may need C and C++ compilers in your search path. Check definitions of `CC` and `CCC` in `base/configure/os/CONFIG.<host>.<host>` or the definitions for `GCC` and `G++` if `ANSI=GCC` and `CPLUSPLUS=GCC` are specified in `CONFIG_SITE`. For building base you also must have `echo` in your search path.

4.2.3.1 Unix path

For Unix host builds you also need `touch`, `cpp`, `cp`, `rm`, `mv`, and `mkdir` in your search path and `/bin/chmod` must exist. On some Unix systems you may also need `ar` and `ranlib` in your path, and the `c` compiler may require `ld` in your path.

4.2.3.2 Win32 PATH

On WIN32 systems, building shared libraries is the default setting and you will need to add `fullpathname to $(INSTALL_LOCATION)/bin/$(EPICS_HOST_ARCH)` to your path. Building shared libraries is determined by the value of the macro `SHARED_LIBRARIES` in `CONFIG_SITE` or `os/CONFIG.Common.<host>` (either YES or NO).

4.2.4 Startup files

The startup directory in EPICS base contains a perl script, `EpicsHostArch.pl`, which can be used to define `EPICS_HOST_ARCH`. This script can be invoked with a command line parameter defining the alternate compiler (e.g. if invoking "`EpicsHostArch.pl`" yields `solaris-sparc`, then invoking "`EpicsHostArch.pl gnu`" will yield `solaris-sparc-gnu`).

The startup directory also contains scripts to help users set the path and other environment variables

4.3 Configuration Definitions

4.3.1 Site-specific EPICS Base Configuration

4.3.1.1 Site configuration

To configure EPICS base for your site, you may want to modify the default definitions in the following files:

configure/CONFIG_SITE Build choices. Specify target archs.
configure/CONFIG_SITE_ENV Environment variable defaults

4.3.1.2 Host configuration

To configure each host system for your site, you may override the default definitions in the configure/os directory by adding a new file with override definitions. The new file should have the same name as the distribution file to be overridden except CONFIG in the name is changed to CONFIG_SITE.

configure/os/CONFIG_SITE.<host>.<host> - Host build settings
configure/os/CONFIG_SITE.<host>.Common - Host build settings for all target systems

4.3.1.3 Target configuration

To configure each target system, you may override the default definitions in the configure/os directory by adding a new file with override definitions. The new file should have the same name as the distribution file to be overridden except CONFIG in the name is replaced by CONFIG_SITE.

configure/os/CONFIG_SITE.Common.<target> - Target cross settings
configure/os/CONFIG_SITE.<host>.<target> - Host-target settings
configure/os/CONFIG_SITE.Common.vxWorksCommon - vxWorks full paths

4.3.1.4 R3.13 compatibility configuration

To configure EPICS base for building with R3.13 extensions and ioc applications, you must modify the default definitions in the base/config/CONFIG_SITE* files to agree with site definitions you made in base/configure and base/configure/os files.

4.3.2 Directory definitions

The configure files contain definitions for locations in which to install various components. These are all relative to INSTALL_LOCATION. The default value for INSTALL_LOCATION is \$(TOP), and \$(T_A) is the current build's target architecture. The default value for INSTALL_LOCATION can be overridden at the bottom of an ioc application's configure/CONFIG file. For base set INSTALL_LOCATION in the CONFIG_SITE file.

```
INSTALL_LOCATION_LIB      = $(INSTALL_LOCATION)/lib
INSTALL_LOCATION_BIN      = $(INSTALL_LOCATION)/bin

INSTALL_HOST_BIN          = $(INSTALL_LOCATION_BIN)/$(EPICS_HOST_ARCH)
INSTALL_HOST_LIB          = $(INSTALL_LOCATION_LIB)/$(EPICS_HOST_ARCH)
```

```

INSTALL_INCLUDE      = $(INSTALL_LOCATION)/include
INSTALL_DOC          = $(INSTALL_LOCATION)/doc
INSTALL_HTML         = $(INSTALL_LOCATION)/html
INSTALL_TEMPLATES    = $(INSTALL_LOCATION)/templates
INSTALL_DBD          = $(INSTALL_LOCATION)/dbd
INSTALL_DB           = $(INSTALL_LOCATION)/db
INSTALL_CONFIG       = $(INSTALL_LOCATION)/configure
INSTALL_JAVA         = $(INSTALL_LOCATION)/javali

INSTALL_LIB          = $(INSTALL_LOCATION_LIB)/$(T_A)
INSTALL_SHRLIB       = $(INSTALL_LOCATION_LIB)/$(T_A)
INSTALL_TCLLIB       = $(INSTALL_LOCATION_LIB)/$(T_A)
INSTALL_BIN          = $(INSTALL_LOCATION_BIN)/$(T_A)

```

4.3.3 Extension and Application Specific Configuration

The base/configure directory contains files with the default build definitions and site specific build definitions. The extensions/configure directory contains extension specific build definitions (e.g. location of X11 and Motif libraries) and "include <filename>" lines for the base/configure files. Likewise, the <application>/configure directory contains application specific build definitions and includes for the base/configure files. Build definitions such a CROSS_COMPILER_TARGET_ARCHS can be overridden in an extension or application by placing an override definition in the <top>/configure/CONFIG_SITE file.

4.3.4 RELEASE file

Every <top>/configure directory contains a RELEASE file. RELEASE contains a user specified list of other <top> directory structures containing files needed by the current <top>. When make is executed in configure, a perl script, convertRelease.pl, generates CONFIG_APP_INCLUDE in the O.<arch> subdirectories which contains include, bin, and library directory definitions for each external <top> definition in the RELEASE file. CONFIG_APP_INCLUDE is included into the CONFIG file so its definitions are available for use by Makefiles. The convertRelease.pl script also generates a RULES_INCLUDE file which contains an include statement for any existing RULES_BUILD files from each external <top> listed in the RELEASE file. RULES_INCLUDE is included by the RULES_BUILD file in EPICS base so all make rules in the external <top> RULES_BUILD files are available for use by Makefiles.

For example, if configure/RELEASE contains the following definition:

```
CAMAC = /home/epics/modules/bus/camac
```

then the created CONFIG_APP_INCLUDE will contain the following lines

```

CAMAC_BIN = /home/epics/modules/bus/camac/bin/solaris-sparc
CAMAC_LIB = /home/epics/modules/bus/camac/lib/solaris-sparc
RELEASE_INCLUDES += -I/home/epics/modules/bus/camac/include
RELEASE_DBDFLAGS += -I /home/epics/modules/bus/camac/dbd

```

and the created RULES_INCLUDE will contain the following line

```
-include /home/epics/modules/bus/camac/configure/RULES_BUILD
```

RELEASE_DBDFLAGS will appear on the command lines for the dbToRecordTypeH, mkmf.pl, and dbExpand tools, and RELEASE_INCLUDES will appear on compiler command lines. CAMAC_LIB and CAMAC_BIN can be used in a Makefile to define the location of needed scripts, executables, object files, libraries or other files.

Definitions in configure/RELEASE can be overridden for a specific EPICS_HOST_ARCH architecture by providing a configure/RELEASE.<epics_host_arch> file containing overriding definitions.

4.3.5 Modifying configure/RELEASE* files

You should always do a "gnumake clean uninstall" in the <top> level directory BEFORE adding, changing, or removing any definitions in the configure/RELEASE* files and then a "gnumake" at the top level AFTER making the changes.

The file <top>/config/RELEASE contains definitions for components obtained from outside <top>. If you want to link to a new release of anything defined in the file do the following:

```
cd <top>
gnumake clean uninstall
edit configure/RELEASE
    change the relevant line(s) to point to the new release
gnumake
```

All definitions in <top>/config/RELEASE must result in complete path definitions, i.e. relative path names are not permitted. If your site could have multiple releases of base and other <supporttop> components installed at once, these path definitions should contain a release number as one of the components. However as the RELEASE file is read by gnumake, it is permissible to use macro substitutions to define these pathnames, for example:

```
SUPPORT = /usr/local/iocapps/R3.13.1
EPICS_BASE = $(SUPPORT)/base/3-13-1-asd2
```

4.3.6 Specifying osclass specific definitions

Definitions in a Makefile will apply to the host system (the platform on which make is executed) and each system defined by CROSS_COMPILER_TARGET_ARCHS.

It is possible to limit the systems for which a particular definition applies. Most Makefile definitions can be specified with an appended underscore "_" followed by an osclass specification. If an _<osclass> is not specified, then the definition applies to the host and all CROSS_COMPILER_TARGET_ARCHS systems. If an _<osclass> is specified, then the definition applies only to systems with the specified os class. A Makefile definition can also have an appended _DEFAULT specification. If _DEFAULT is appended, then the Makefile definition will apply to all systems that do not have an _<osclass> specification for that definition. If a _DEFAULT is appended on a definition but the definition should not apply to a particular system <osclass>, the value "-nil-" should be specified in the relevant Makefile definition.

Each system has an OS_CLASS definition in its configure/os/CONFIG.Common.<arch> file. A few examples are:

```
For vxWorks-* targets <osclass> is vxWorks.
For RTEMS-* targets <osclass> is RTEMS.
For solaris-* targets <osclass> is solaris.
For win32-* targets <osclass> is WIN32.
For linux-* targets <osclass> is Linux.
For hpux-* targets <osclass> is hpux.
For darwin-* targets <osclass> is Darwin.
For aix-* targets <osclass> is AIX.
```

For example the following Makefile lines specify that product aaa should be created for all systems. Product bbb should be created for systems that do not have OS_CLASS defined as solaris.

```
PROD = aaa
PROD_solaris = -nil-
PROD_DEFAULT = bbb
```

4.3.7 Specifying T_A specific definitions

It is possible for the user to limit the systems for which a particular definition applies to specific target systems.

For example the following Makefile lines specify that product aaa should be created for all target architecture which allow IOC type products and product bbb should be created only for the vxWorks-68040 and vxWorks-ppc603 targets. Remember T_A is the build's current target architecture. so PROD_IOC has the bbb value only when the current built target architecture is vxWorks-68040 or vxWorks-ppc603

```
PROD_IOC = aaa
VX_PROD_vxWorks-68040 = bbb
VX_PROD_vxWorks-ppc603 = bbb
PROD_IOC += VX_PROD_$(T_A)
```

4.3.8 Host and Ioc targets

Build creates two type of makefile targets: Host and Ioc. Host targets are executables, object files, libraries, and scripts which are not part of iocCore. Ioc targets are components of ioc libraries, executables, object files, or iocsh scripts which will be run on an ioc.

Each supported target system has a VALID_BUILDS definition which specifies the type of makefile targets it can support. This definition appears in configure/os/CONFIG.Common.<arch> or configure/os/CONFIG.<arch>.<arch> files.

For vxWorks systems VALID_BUILDS is set to "Ioc".

For Unix type systems, VALID_BUILDS is set to "Host Ioc".

For RTEMS systems, VALID_BUILDS is set to "Ioc".

For WIN32 systems, VALID_BUILDS is set to "Host Ioc".

In a Makefile it is possible to limit the systems for which a particular PROD, TESTPROD, LIBRARY, SCRIPTS, and OBJS is built. For example the following Makefile lines specify that product aaa should be created for systems that support Host type builds. Product bbb should be created for systems that support Ioc type builds. Product ccc should be created for all target systems.

```
PROD_HOST = aaa
PROD_IOC = bbb
PROD = ccc
```

These definitions can be further limited by specifying an appended underscore "_" followed by an osclass or DEFAULT specification.

4.4 Makefiles

4.4.1 Name

The name of the makefile in each directory must be Makefile.

4.4.2 Included Files

Makefiles normally include files from <top>/configure. Thus the makefile "inherits" rules and definitions from configure. The files in <top>/configure may in turn include files from another <top>/configure. This technique makes it possible to share make variables and even rules across <top> directories.

4.4.3 Contents of Makefiles

4.4.3.1 Makefiles in directories containing subdirectories

A Makefile in this type of directory must define where <top> is relative to this directory, include <top>/configure files, and specify the subdirectories in the desired order of make execution. Running gnumake in a directory with the following Makefile lines will cause gnumake to be executed in <dir1> first and then <dir2>. The build rules do not allow a Makefile to specify both subdirectories and components to be built.

```
TOP=../..
include $(TOP)/configure/CONFIG
DIRS += <dir1> <dir2>
include $(TOP)/configure/RULES_DIRS
```

4.4.3.2 Makefiles in directories where components are to be built

A Makefile in this type of directory must define where <top> is relative to this directory, include <top> configure files, and specify the target component definitions. Optionally it may contain user defined rules. Running gnumake in a directory with this type of Makefile will cause gnumake to create an O.<arch> subdirectory and then execute gnumake to build the defined components in this subdirectory. It contains the following lines:

```
TOP=../../..
include $(TOP)/configure/CONFIG
    <component definition lines>
include $(TOP)/configure/RULES
    <optional rules definitions>
```

4.4.4 Simple Makefile examples

Create an IOC type library named asIoc from the source file asDbLib.c and install it into the \$(INSTALL_LOCATION)/lib/<arch> directory.

```
TOP=../../..
include $(TOP)/configure/CONFIG
LIBRARY_IOC += asIoc
asIoc_SRCS += asDbLib.c
include $(TOP)/configure/RULES
```

For each Host type target architecture, create an executable named catest from the catest1.c and catest2.c source files linking with the existing EPICS base ca and Com libraries, and then install the catest executable into the \$(INSTALL_LOCATION)/bin/<arch> directory.

```
TOP=../../..
include $(TOP)/configure/CONFIG
PROD_HOST = catest
catest_SRCS += catest1.c catest2.c
catest_LIBS = ca Com
```

```
include $(TOP)/configure/RULES
```

4.5 Make

4.5.1 Make vs. gnumake

EPICS provides an extensive set of make rules. These rules only work with the GNU version of make, gnumake, which is supplied by the Free Software Foundation. Thus, on most Unix systems, the native make will not work. On some systems, e.g. Linux, GNU make may be the default. This manual always uses gnumake in the examples.

4.5.2 Frequently used Make commands

NOTE: It is possible to invoke the following commands for a single target architecture by appending `.<arch>` to the target in the command.

The most frequently used make commands are:

gnumake

This rebuilds and installs everything that is not up to date.

NOTE: Executing gnumake without arguments is the same as "gnumake install"

gnumake <arch>

This rebuilds and installs everything that is not up to date for a single specified target arch.

NOTE: This is the same as "gnumake install.<arch>"

gnumake clean

This can be used to save disk space by deleting the `O.<arch>` directories that gnumake will create, but does not remove any installed files from the `bin`, `db`, `dbd` etc. directories. "gnumake clean.<arch>" can be invoked to clean a single architecture.

gnumake realclean

This command can be executed from the `<top>` directory only. It will remove ALL the `O.<arch>` subdirectories (even those created by a gnumake from another `EPICS_HOST_ARCH`).

gnumake rebuild

This is the same as "gnumake clean install". If you are unsure about the state of the generated files in an application, just execute "gnumake rebuild".

gnumake uninstall

This command can be executed from the `<top>` directory only. It will remove everything installed by gnumake in the `include`, `lib`, `bin`, `db`, `dbd`, etc. directories.

gnumake realuninstall

This command can be executed from the `<top>` directory only. It will remove all the install directories, `include`, `lib`, `bin`, `db`, `dbd`, etc..

gnumake distclean

This command can be executed from the `<top>` directory only. It is the same as issuing both the `realclean` and `realuninstall` commands

gnumake archclean

This command will remove O.<arch> dirs but not O.Common dir.

gnumake cvsclean

This command can be executed from the <top> directory only. It removes cvs .* files in all dirs of directory tree.

4.5.3 Make targets

The following is a summary of targets that can be specified for gnumake:

- <action>
- <arch>
- <action>.<arch>
- <dir>
- <dir>.<action>
- <dir>.<arch>
- <dir>.<action>.<arch>

where:

<arch> is EPICS_HOST_ARCH, solaris-sparc, vxWorks-68040, win32-x86, etc. - builds named architecture only.

<action> is help, clean, realclean, distclean, inc, install, build, rebuild, buildInstall, realuninstall, or uninstall

NOTE: help, uninstall, distclean, and realuninstall can only be specified at <top>.

NOTE: realclean cannot be specified in an *.<arch> subdirectory.

<dir> is subdirectory name

Note: You can build using your os vendors' native compiler and also build using a supported alternate compiler in the same directory structure because the executables and libraries will be created and installed into separate directories (e.g bin/solaris-sparc and bin/solaris-sparc-gnu). You can do this by changing your EPICS_HOST_ARCH, environment variable between builds or by setting EPICS_HOST_ARCH on the gnumake command line.

4.5.4 Header file dependencies

All product, test product, and library source files which appear in one of the source file definitions (e.g. SRCS, PROD_SRCS, LIB_SRCS, <prodname>_SRCS) will have their header file dependencies automatically generated and included as part of the Makefile if HDEPENDS is set to YES in the Makefile and/or in base/configure/CONFIG_SITE.

4.6 Makefile definitions

The following components can be defined in a Makefile:

4.6.1 Source file directories

Normally all product, test product, and library source files reside in the same directory as the Makefile. OS specific source files are allowed and should reside in subdirectories os/<os_class> or os/posix or os/default.

The build rules also allow source files to reside in subdirectories of the current Makefile directory (src directory). For each subdirectory <dir> containing source files add the SRC_DIRS definition.

```
SRC_DIRS += <dir>
```

where <dir> is a relative path definition. An example of SRC_DIRS is

```
SRC_DIRS += ../dir1 ../dir2
```

The directory search order for the above definition is

```
.
../os/$(OS_CLASS) ../os/posix ../os/default
../dir1/os/$(OS_CLASS) ../dir1/os/posix ../dir1/os/default
../dir2/os/$(OS_CLASS) ../dir2/os/posix ../dir2/os/default
..
../dir1 ../dir2
```

where the build directory O.<os_class> is . and the src directory is ...

4.6.2 Posix C source code

The epics base config files assume posix source code and define POSIX to be YES as the default. Individual Makefiles can override this by setting POSIX to NO.

4.6.3 Breakpoint Tables

For each breakpoint table dbd file, bpt<table name>.dbd, to be created from an existing bpt<table name>.data file, add the definition

```
DBD += bpt<table name>.dbd
```

to the Makefile. The following Makefile will create a bptTypeJdegC.dbd file from an existing bptTypeJdegC.data file and install the new dbd file into the \$(INSTALL_LOCATION)/dbd directory.

```
TOP=../../..
include $(TOP)/configure/CONFIG
DBD += bptTypeJdegC.dbd
include $(TOP)/configure/RULES
```

4.6.4 Record Type Definitions

For each new record type, the following definition should be added to the makefile:

```
DBDINC += <rectype>Record
```

A <rectype>Record.h header file will be created from an existing <rectype>Record.dbd file. This header will be installed into the \$(INSTALL_LOCATION)/include directory and the dbd file will be installed into the \$(INSTALL_LOCATION)/dbd directory.

The following Makefile will create xxxRecord.h from an existing xxxRecord.dbd file, install xxxRecord.h into \$(INSTALL_LOCATION)/include, and install xxxRecord.dbd into \$(INSTALL_LOCATION)/dbd.

```
TOP=../../..
include $(TOP)/configure/CONFIG
DBDINC += xxxRecord
include $(TOP)/configure/RULES
```

4.6.5 Menus

If a menu `menu<name>.dbd` file is present, then add the following definition:

```
DBDINC += menu<name>.h
```

The header file, `menu<name>.h` will be created from the existing `menu<name>.dbd` file and installed into the `$(INSTALL_LOCATION)/include` directory and the menu dbd file will be installed into `$(INSTALL_LOCATION)/dbd`.

The following Makefile will create a `menuConvert.h` file from an existing `menuConvert.dbd` file and install `menuConvert.h` into `$(INSTALL_LOCATION)/include` and `menuConvert.dbd` into `$(INSTALL_LOCATION)/dbd`.

```
TOP=../../../../..
include $(TOP)/configure/CONFIG
DBDINC = menuConvert.h
include $(TOP)/configure/RULES
```

4.6.6 Expanded Database Definition Files

Database definition include files named `<name>Include.dbd` containing includes for other database definition files can be expanded by the utility program `dbExpand` into a created `<name>.dbd` file and the `<name>.dbd` file installed into `$(INSTALL_LOCATION)/dbd`. The following variables control the process:

```
DBD += <name>.dbd
USR_DBDFLAGS += -I <include path>
USR_DBDFLAGS += -S <macro substitutions>
<name>_DBD += <file1>.dbd <file2>.dbd ...
```

where

```
DBD += <name>.dbd
```

is the name of the output dbd file to contain the expanded definitions. It is created by expanding an existing or build created `<name>Include.dbd` file and then copied into `$(INSTALL_LOCATION)/dbd`.

An example of a file to be expanded is `exampleInclude.dbd` containing the following lines

```
include "base.dbd"
include "xxxRecord.dbd"
device(xxx, CONSTANT, devXxxSoft, "SoftChannel")
```

`USR_DBDFLAGS` defines optional flags for `dbExpand`. Currently only an include path (`-I <path>`) and macro substitution (`-S <substitution>`) are supported. The include paths for EPICS `base/dbd`, and other `<top>/dbd` directories will automatically be added during the build if the `<top>` names are specified in the `configure/RELEASE` file.

A database definition include file named `<name>Include.dbd` containing includes for other database definition files can be created from a `<name>_DBD` definition. The lines

```
DBD += <name>.dbd
<name>_DBD += <file1>.dbd <file2>.dbd ...
```

will create an expanded dbd file `<name>.dbd` by first creating a `<name>Include.dbd`. For each filename in the `<name>_DBD` definition, the created `<name>Include.dbd` will contain an include statement for that filename. Then the expanded DBD file is generated from the created `<name>Include.dbd` file and installed into `$(INSTALL_LOCATION)/dbd`.

The following Makefile will create an expanded dbd file named `example.dbd` from an existing `exampleInclude.dbd` file and then install `example.dbd` into the `$(INSTALL_LOCATION)/dbd` directory.

```
TOP=../../../../..
```

```
include $(TOP)/configure/CONFIG
DBD += exampleApp.dbd
include $(TOP)/configure/RULES
```

The following Makefile will create an exampleInclude.dbd file from the example_DBD definition then expand it to create an expanded dbd file, example.dbd, and install example.dbd into the \$(INSTALL_LOCATION)/dbd directory.

```
TOP=../../..
include $(TOP)/configure/CONFIG
DBD += example.dbd
example_DBD += base.dbd xxxRecord.dbd xxxSupport.dbd
include $(TOP)/configure/RULES
```

The created exampleInclude.dbd file will contain the following lines

```
include "base.dbd"
include "xxxRecord.dbd"
include "xxxSupport.dbd"
```

4.6.7 Registering Support Routines for Expanded Database Definition Files

A source file which registers simple static variables and record/device/driver support routines with iocsh can be created. The list of variables and routines to register is obtained from lines in an existing dbd file.

The following line in a Makefile will result in <name>_registerRecordDeviceDriver.cpp being created, compiled, and linked into <prodname>. It requires that the file <name>.dbd exist.

```
<prodname>_SRCS += <name>_registerRecordDeviceDriver.cpp
```

An example of registering the variable mySubDebug and the routines mySubInit and mySubProcess is <name>.dbd containing the following lines

```
variable(mySubDebug)
function(mySubInit)
function(mySubProcess)
```

4.6.8 Database Definition Files

The following line installs the existing named dbd files into \$(INSTALL_LOCATION)/dbd without expansion.

```
DBD += <name>.dbd
```

4.6.9 DBD install files

Definitions of the form:

```
DBD_INSTALLS += <name>
```

result in files being installed to the \$(INSTALL_LOCATION)/dbd directory. The file <name> can appear with or without a directory prefix. If the file has a directory prefix e.g. \$(APPNAME)/dbd/, it is copied from the specified location. If a directory prefix is not present, make will look in the current source directory for the file.

4.6.10 Database Files

For most databases just the name of the database has to be specified. Make will figure out how to generate the file:

```
DB += xxx.db
```

generates xxx.db depending on which source files exist and installs it into \$(INSTALL_LOCATION)/db.

A <name>.db database file will be created from an optional <name>.template file and/or an optional <name>.substitutions file, If the substitution file exists but the template file is not named <name>.template, the template file name can be specified as

```
<name>_TEMPLATE = <template file name>
```

A *<nn>.db database file will be created from a *.template and a *<nn>.substitutions file, (where nn is an optional index number).

If a <name> substitutions file contains "file" references to other input files, these referenced files should be made dependencies of the created <name>.db by adding a dependency definition line:

```
<name>_DEPENDS = <filename1> <filename2> . . .
```

The Macro Substitutions and Include tool, msi, will be used to generate the database, and msi must either be in your path or you must redefine MSI as the full path name to the msi binary in a RELEASE file or Makefile. An example MSI definition is

```
MSI = /usr/local/epics/extensions/bin/${EPICS_HOST_ARCH}/msi
```

Template files <name>.template, and db files, <name>.db, will be created from an edf file <name>.edf and an <name>.edf file will be created from a <name>.sch file.

Template and substitution files can be installed.

```
DB += xxx.template xxx.substitutions
```

generates and installs these files. If one or more xxx.substitutions files are to be created by script, the script name must be placed in the CREATSUBSTITUTIONS variable (e.g. CREATSUBSTITUTIONS=mySubst.pl). This script will be executed by gnumake with the prefix of the substitution file name to be generated as its argument. If (and only if) there are script generated substitutions files, the prefix of any inflated database's name may not equal the prefix of the name of any template used within the directory.

4.6.11 DB install files

Definitions of the form:

```
DB_INSTALLS += <name>
```

result in files being installed to the \$(INSTALL_LOCATION)/db directory. The file <name> can appear with or without a directory prefix. If the file has a directory prefix e.g. \$(APPNAME)/db/, it is copied from the specified location. If a directory prefix is not present, make will look in the current source directory for the file.

4.6.12 Compile and link command options

Any of the following can be specified:

4.6.12.1 Options for all compile/link commands.

These definitions will apply to all compiler and linker targets.

```
USR_INCLUDES += -I<name>  
    header file directories each prefixed by a "-I".  
USR_INCLUDES_<osclass> += -I<name>  
    os specific header file directories each prefixed by a "-I".
```

```

USR_INCLUDES_DEFAULT += -I<name>
    header file directories each prefixed by a "-I" for any arch that does not have a USR_INCLUDE_<osclass>
    definition
USR_CFLAGS += <c flags>
    c compiler options.
USR_CFLAGS_<osclass> += <c flags>
    os specific c compiler options.
USR_CFLAGS_<T_A> += <c flags>
    target architecture specific c compiler options.
USR_CFLAGS_DEFAULT += <c flags>
    c compiler options for any arch that does not have a USR_CFLAGS_<osclass> definition
USR_CXXFLAGS += <c++ flags>
    c++ compiler options.
USR_CXXFLAGS_<osclass> += <c++ flags>
    c++ compiler options for the specified osclass.
USR_CXXFLAGS_<T_A> += <c++ flags>
    c++ compiler options for the specified target architecture.
USR_CXXFLAGS_DEFAULT += <c++ flags>
    c++ compiler options for any arch that does not have a USR_CXXFLAGS_<osclass> definition
USR_CPPFLAGS += <preprocessor flags>
    c preprocessor options.
USR_CPPFLAGS_<osclass> += <preprocessor flags>
    os specific c preprocessor options.
USR_CPPFLAGS_<T_A> += <preprocessor flags>
    target architecture specific c preprocessor options.
USR_CPPFLAGS_DEFAULT += <preprocessor flags>
    c preprocessor options for any arch that does not have a USR_CPPFLAGS_<osclass> definition
USR_LDFLAGS += <linker flags>
    linker options.
USR_LDFLAGS_<osclass> += <linker flags>
    os specific linker options.
USR_LDFLAGS_DEFAULT += <linker flags>
    linker options for any arch that does not have a USR_LDFLAGS_<osclass> definition
  
```

4.6.12.2 Options for a target specific compile/link command.

```

<name>_INCLUDES += -I<name>
    header file directories each prefixed by a "-I".
<name>_INCLUDES_<osclass> += -I<name>
    os specific header file directories each prefixed by a "-I".
<name>_INCLUDES_<T_A> += -I<name>
    target architecture specific header file directories each prefixed by a "-I".
<name>_CFLAGS += <c flags>
    c compiler options.
<name>_CFLAGS_<osclass> += <c flags>
    os specific c compiler options.
<name>_CFLAGS_<T_A> += <c flags>
    target architecture specific c compiler options.
<name>_CXXFLAGS += <c++ flags>
    c++ compiler options.
<name>_CXXFLAGS_<osclass> += <c++ flags>
    c++ compiler options for the specified osclass.
<name>_CXXFLAGS_<T_A> += <c++ flags>
  
```

++ compiler options for the specified target architecture.
`<name>_CPPFLAGS += <preprocessor flags>`
 c preprocessor options.
`<name>_CPPFLAGS_<osclass> += <preprocessor flags>`
 os specific c preprocessor options.
`<name>_CPPFLAGS_<T_A> += <preprocessor flags>`
 target architecture specific c preprocessor options.
`<name>_LDFLAGS += <linker flags>`
 linker options.
`<name>_LDFLAGS_<osclass> += <linker flags>`
 os specific linker options.

4.6.13 Libraries

A library is created and installed into `$(INSTALL_LOCATION)/lib/<arch>` by specifying its name and the name of the object and/or source files containing code for the library. An object or source file name can appear with or without a directory prefix. If the file name has a directory prefix e.g. `$(EPICS_BASE_BIN)`, it is taken from the specified location. If a directory prefix is not present, make will first look in the source directories for a file with the specified name and next try to create the file using existing configure rules. A library filename prefix may be prepended to the library name when the file is created. For Unix type systems and vxWorks the library prefix is `lib` and there is no prefix for WIN32. Also a library suffix appropriate for the library type and target arch (e.g. `.a`, `.so`, `.lib`, `.dll`) will be appended to the filename when the file is created.

vxWorks and RTEMS Note: Only archive libraries are created.

Shared libraries Note: Shared libraries can be built for any or all HOST type architectures. The definition of `SHARED_LIBRARIES` (YES/NO) in `base/configure/CONFIG_SITE` determines whether shared or archive libraries will be built. When `SHARED_LIBRARIES` is YES, both archive and shared libraries are built. This definition can be overridden for a specific arch in an `configure/os/CONFIG_SITE.<arch>.Common` file. The default definition for `SHARED_LIBRARIES` in the EPICS base distribution file is YES for all host systems.

win32 Note: An object library file is created when `SHARED_LIBRARIES=NO`, `<name>.lib` which is installed into `$(INSTALL_LOCATION)/lib/<arch>`. Two library files are created when `SHARED_LIBRARIES=YES`, `<name>.lib` which is installed into `$(INSTALL_LOCATION)/lib/<arch>`, and `<name>.dll` which is installed into `$(INSTALL_LOCATION)/bin/<arch>`. (Warning: The file `<name>.lib` will only be created by the build if there are exported symbols from the library.) If `SHARED_LIBRARIES=YES`, the directory `$(INSTALL_LOCATION)/bin/<arch>` must be in the user's path during builds to allow invoking executables which were linked with shared libraries.

4.6.13.1 Specifying the library name.

Any of the following can be specified:

`LIBRARY += <name>`
 A library will be created for every target arch.
`LIBRARY_<osclass> += <name>`
 Library `<name>` will be created for all archs of the specified `osclass`.
`LIBRARY_DEFAULT += <name>`
 Library `<name>` will be created for any arch that does not have a `LIBRARY_<osclass>` definition
`LIBRARY_IOC += <name>`
 Library `<name>` will be created for IOC type archs.
`LIBRARY_IOC_<osclass> += <name>`
 Library `<name>` will be created for all IOC type archs of the specified `osclass`.
`LIBRARY_IOC_DEFAULT += <name>`

Library <name> will be created for any IOC type arch that does not have a LIBRARY_IOC_<osclass> definition

LIBRARY_HOST += <name>
Library <name> will be created for HOST type archs.

LIBRARY_HOST_<osclass> += <name>
Library <name> will be created for all HOST type archs of the specified osclass.

LIBRARY_HOST_DEFAULT += <name>
Library <name> will be created for any HOST type arch that does not have a LIBRARY_HOST_<osclass> definition

4.6.13.2 Specifying library source file names

Source file names, which must have a suffix, are defined as follows:

SRCS += <name>
Source files will be used for all defined libraries and products.

SRCS_<osclass> += <name>
Source files will be used for all defined libraries and products for all archs of the specified osclass.

SRCS_DEFAULT += <name>
Source files will be used for all defined libraries and products for any arch that does not have a SRCS_<osclass> definition

LIBSRCS and LIB_SRCS have the same meaning. LIBSRCS is deprecated, but retained for R3.13 compatibility.

LIBSRCS += <name>
Source files will be used for all defined libraries.

LIBSRCS_<osclass> += <name>
Source files will be used for all defined libraries for all archs of the specified osclass.

LIBSRCS_DEFAULT += <name>
Source files will be used for all defined libraries for any arch that does not have a LIBSRCS_<osclass> definition

USR_SRCS += <name>
Source files will be used for all defined products and libraries.

USR_SRCS_<osclass> += <name>
Source files will be used for all defined products and libraries for all archs of the specified osclass.

USR_SRCS_DEFAULT += <name>
Source files will be used for all defined products and libraries for any arch that does not have a USR_SRCS_<osclass> definition

LIB_SRCS += <name>
Source files will be used for all libraries.

LIB_SRCS_<osclass> += <name>
Source files will be used for all defined libraries for all archs of the specified osclass.

LIB_SRCS_DEFAULT += <name>
Source files will be used for all defined libraries for any arch that does not have a LIB_SRCS_<osclass> definition

<libname>_SRCS += <name>
Source files will be used for the named library.

<libname>_SRCS_<osclass> += <name>
Source files will be used for named library for all archs of the specified osclass.

<libname>_SRCS_DEFAULT += <name>
Source files will be used for named library for any arch that does not have a <libname>_SRCS_<osclass> definition

4.6.13.3 Specifying library object file names

Library object file names should only be specified for object files which will not be built in the current directory. For object files built in the current directory, library source file names should be specified. See Specifying Library Source File Names above.

Object files which have filename with a ".o" or ".obj" suffix are defined as follows and can be specified without the suffix but should have the directory prefix

```
USR_OBJS += <name>
    Object files will be used in builds of all products and libraries
USR_OBJS_<osclass> += <name>
    Object files will be used in builds of all products and libraries for archs with the specified osclass.
USR_OBJS_DEFAULT += <name>
    Object files will be used in builds of all products and libraries for archs without a USR_OBJS_<osclass>
    definition specified.
LIB_OBJS += <name>
    Object files will be used in builds of all libraries.
LIB_OBJS_<osclass> += <name>
    Object files will be used in builds of all libraries for archs of the specified osclass.
LIB_OBJS_DEFAULT += <name>
    Object files will be used in builds of all libraries for archs without a LIB_OBJS_<osclass> definition
    specified.
<libname>_OBJS += <name>
    Object files will be used for all builds of the named library)
<libname>_OBJS_<osclass> += <name>
    Object files will be used in builds of the library for archs with the specified osclass.
<libname>_OBJS_DEFAULT += <name>
    Object files will be used in builds of the library for archs without a <libname>_OBJS_<osclass> definition
    specified.
```

Combined object files, from R3.13 built modules and applications which have file names that do not include a ".o" or ".obj" suffix (e.g. xyzLib) are defined as follows:

```
USR_OBGLIBS += <name>
    Combined object files will be used in builds of all libraries and products.
USR_OBGLIBS_<osclass> += <name>
    Combined object files will be used in builds of all libraries and products for archs of the specified osclass.
USR_OBGLIBS_DEFAULT += <name>
    Combined object files will be used in builds of all libraries and products for archs without a
    USR_OBGLIBS_<osclass> definition specified.

LIB_OBGLIBS += <name>
    Combined object files will be used in builds of all libraries.
LIB_OBGLIBS_<osclass> += <name>
    Combined object files will be used in builds of all libraries for archs of the specified osclass.
LIB_OBGLIBS_DEFAULT += <name>
    Combined object files will be used in builds of all libraries for archs without a LIB_OBGLIBS_<osclass>
    definition specified.

<libname>_OBGLIBS += <name>
    Combined object files will be used for all builds of the named library.
<libname>_OBGLIBS_<osclass> += <name>
    Combined object files will be used in builds of the library for archs with the specified osclass.
```

```
<libname>_OBJLIBS_DEFAULT += <name>
    Combined object files will be used in builds of the library for archs without a
    <libname>_OBJLIBS_<osclass> definition specified.
```

```
<libname>_LDOBJs += <name>
    Combined object files will be used for all builds of the named library. (deprecated)
```

```
<libname>_LDOBJs_<osclass> += <name>
    Combined object files will be used in builds of the library for archs with the specified osclass. (deprecated)
```

```
<libname>_LDOBJs_DEFAULT += <name>
    Combined object files will be used in builds of the library for archs without a <libname>_OBJs_<osclass>
    definition specified. (deprecated)
```

4.6.13.4 LIBOBJs definitions

Previous versions of epics (3.13 and before) accepted definitions like:

```
LIBOBJs += $(<support>_BIN)/xxx.o
```

These are gathered together in files such as baseLIBOBJs. To use such definitions include the lines:

```
-include ../baseLIBOBJs
<libname>_OBJs += $(LIBOBJs)
```

Note: vxWorks applications created by makeBaseApp.pl from base release R3.14.0alpha3 and later no longer have a file named baseLIBOBJs. Base record and device support now exists in archive libraries.

4.6.13.5 Specifying dependant libraries to be linked when creating a library

For each library name specified which is not a system library nor a library from an EPICS top defined in the config/re/RELEASE file, a <name>_DIR definition must be present in the Makefile to specify the location of the library.

Library names, which must not have a directory and "lib" prefix nor a suffix, are defined as follows:

```
LIB_LIBS += <name>
    Libraries to be used when linking all defined libraries.
```

```
LIB_LIBS_<osclass> += <name>
    Libraries to be used or all archs of the specified osclass when linking all defined libraries.
```

```
LIB_LIBS_DEFAULT += <name>
    Libraries to be used for any arch that does not have a LIB_LIBS_<osclass> definition when linking all
    defined libraries.
```

```
USR_LIBS += <name>
    Libraries to be used when linking all defined products and libraries.
```

```
USR_LIBS_<osclass> += <name>
    Libraries to be used or all archs of the specified osclass when linking all defined products and libraries.
```

```
USR_LIBS_DEFAULT += <name>
    Libraries to be used for any arch that does not have a USR_LIBS_<osclass> definition when linking all
    defined products and libraries.
```

```
<libname>_LIBS += <name>
    Libraries to be used for linking the named library.
```

```
<libname>_LIBS_<osclass> += <name>
    Libraries will be used for all archs of the specified osclass for linking named library.
```

```
<libname>_LIBS_DEFAULT += <name>
    Libraries to be used for any arch that does not have a <libname>_LIBS_<osclass> definition when linking
    named library.
```

```
<libname>_SYS_LIBS += <name>
```

System libraries to be used for linking the named library.

```
<libname>_SYS_LIBS_<osclass> += <name>
```

System libraries will be used for all archs of the specified osclass for linking named library.

```
<libname>_SYS_LIBS_DEFAULT += <name>
```

System libraries to be used for any arch that does not have a <libname>_LIBS_<osclass> definition when linking named library.

4.6.13.6 The order of dependant libraries

Dependant library names appear in the following order on a library link line:

1. <libname>_LIBS
2. <libname>_LIBS_<osclass> or <libname>_LIBS_DEFAULT
3. LIB_LIBS
4. LIB_LIBS_<osclass> or LIB_LIBS_DEFAULT
5. USR_LIBS
6. USR_LIBS_<osclass> or USR_LIBS_DEFAULT
7. <libname>_SYS_LIBS
8. <libname>_SYS_LIBS_<osclass> or <libname>_SYS_LIBS_DEFAULT
9. LIB_SYS_LIBS
10. LIB_SYS_LIBS_<osclass> or LIB_SYS_LIBS_DEFAULT
11. USR_SYS_LIBS
12. USR_SYS_LIBS_<osclass> or USR_SYS_LIBS_DEFAULT

4.6.13.7 Specifying library DLL file names (deprecated)

WIN32 libraries require all external references to be resolved, so if a library contains references to items in other DLL libraries, these DLL library names must be specified (without directory prefix and without ".dll" suffix) as follows:

```
DLL_LIBS += <name>
```

These DLLs will be used for all libraries.

```
<libname>_DLL_LIBS += <name>
```

These DLLs will be used for the named library.

Each <name> must have a corresponding <name>_DIR definition specifying its directory location.

4.6.13.8 Specifying shared library version number

A library version number can be specified when creating a shared library as follows:

```
SHRLIB_VERSION += <version>
```

On WIN32 this results in "/version:\$(SHRLIB_VERSION)" link option. On Unix type hosts ".\$(SHRLIB_VERSION)" is appended to the shared library name and a symbolic link is created for the unversioned library name. \$(EPICS_VERSION).\$(EPICS_REVISION) is the default value for SHRLIB_VERSION.

4.6.13.9 Library example:

```
LIBRARY_vxWorks += vxWorksOnly  
LIBRARY_IOC += iocOnly  
LIBRARY_HOST += hostOnly  
LIBRARY += all
```

```

vxWorksOnly_OBJS += $(LINAC_BIN)/vxOnly1
vxWorksOnly_SRCS += vxOnly2.c
iocOnly_OBJS += $(LINAC_BIN)/iocOnly1
iocOnly_SRCS += iocOnly2.cpp
hostOnly_OBJS += $(LINAC_BIN)/host1
all_OBJS += $(LINAC_BIN)/all1
all_SRCS += all2.cpp

```

If the architectures defined in <top>/configure are solaris-sparc and vxWorks-68040 and LINAC is defined in the <top>/CONFIGURE/RELEASE file, then the following libraries will be created:

- \$(INSTALL_LOCATION)/bin/vxWork-68040/libvxWorksOnly.a : \$(LINAC_BIN)/vxOnly1.o vxOnly2.o
- \$(INSTALL_LOCATION)/bin/vxWork-68040/libiocOnly.a : \$(LINAC_BIN)/iocOnly1.o iocOnly2.o
- \$(INSTALL_LOCATION)/lib/solaris-sparc/libiocOnly.a : \$(LINAC_BIN)/iocOnly1.o iocOnly2.o
- \$(INSTALL_LOCATION)/lib/solaris-sparc/libhostOnly.a : \$(LINAC_BIN)/host1.o
- \$(INSTALL_LOCATION)/bin/vxWork-68040/liball.a : \$(LINAC_BIN)/all1.o all2.o
- \$(INSTALL_LOCATION)/lib/solaris-sparc/liball.a : \$(LINAC_BIN)/all1.o all2.o

4.6.14 Loadable libraries

Loadable libraries are regular libraries which are not required to have all symbols resolved during the build. The intent is to create dynamic plugins so no archive library is created. Source file, object files, and dependant libraries are specified in exactly the same way as for regular libraries.

Any of the following can be specified:

```
LOADABLE_LIBRARY += <name>
```

The <name> loadable library will be created for every target arch.

```
LOADABLE_LIBRARY_<osclass> += <name>
```

Loadable library <name> will be created for all archs of the specified osclass.

```
LOADABLE_LIBRARY_DEFAULT += <name>
```

Loadable library <name> will be created for any arch that does not have a LOADABLE_LIBRARY_<osclass> definition

```
LOADABLE_LIBRARY_HOST += <name>
```

Loadable library <name> will be created for HOST type archs.

```
LOADABLE_LIBRARY_HOST_<osclass> += <name>
```

Loadable library <name> will be created for all HOST type archs of the specified osclass.

```
LOADABLE_LIBRARY_HOST_DEFAULT += <name>
```

Loadable library <name> will be created for any HOST type arch that does not have a LOADABLE_LIBRARY_HOST_<osclass> definition

4.6.15 Combined object libraries (VxWorks only)

Combined object libraries are regular combined object files which have been created by linking together multiple object files. OBJLIB specifications in the Makefile create a combined object file and a corresponding munch file for vxWorks target architectures only. Combined object libraries have a Library.o suffix. It is possible to generate and install combined object libraries by using definitions:

```

OBJLIB += <name>
OBJLIB_vxWorks += <name>
OBJLIB_SRCS += <srcname1> <srcname2> ...
OBJLIB_OBJS += <objname1> <objname2> ...

```

These definitions result in the combined object file <name>Library.o and its corresponding <name>Library.munch munch file being built for each vxWorks architecture from source/object files in the OBJLIB_SRCS/OBJLIB_OBJS definitions. The combined object file and the munch file are installed into the \$(INSTALL_LOCATION)/bin/<arch> directory.

4.6.16 Object Files

It is possible to generate and install object files by using definitions:

```
OBJS += <name>
OBJS_<osclass> += <name>
OBJS_DEFAULT += <name>
OBJS_IOC += <name>
OBJS_IOC_<osclass> += <name>
OBJS_IOC_DEFAULT += <name>
OBJS_HOST += <name>
OBJS_HOST_<osclass> += <name>
OBJS_HOST_DEFAULT += <name>
```

These will cause the specified file to be generated from an existing source file for the appropriate target arch and installed into \$(INSTALL_LOCATION)/bin/<target_arch>.

The following Makefile will create the abc object file for all target architectures, the def object file for all target archs except vxWorks, and the xyz object file only for the vxWorks target architecture and install them into the appropriate \$(INSTALL_LOCATION)/bin/<arch> directory.

```
TOP=../../../../
include $(TOP)/configure/CONFIG
OBJS += abc
OBJS_vxWorks += xyz
OBJS_DEFAULT += def
include $(TOP)/configure/RULES
```

4.6.17 State Notation Programs

A state notation program file can be specified as a source file in any SRC definition. For example:

```
<prodname>_SRCS += <name>.stt
```

The state notation compiler `snc` will generate the file <name>.c from the state notation program file <name>.stt. This C file is compiled and the resulting object file is linked into the <prodname> product.

A state notation source file must have the extension `.st` or `.stt`. The `.st` file is passed through the C preprocessor before it is processed by `snc`.

If you have state notation language source files (`.stt` and `.st` files), the module `seq` must be built and `SNCSEQ` defined in the `RELEASE` file. If the state notation language source files require c preprocessing before conversion to c source (`.st` files), `gcc` must be in your path.

4.6.18 Scripts, etc.

Any of the following can be specified:

```
SCRIPTS += <name>
    A script will be installed from the src directory to the $(INSTALL_LOCATION)/bin/<arch> directories.
SCRIPTS_<osclass> += <name>
```

Script <name> will be installed for all archs of the specified osclass.
`SCRIPTS_DEFAULT += <name>`
 Script <name> will be installed for any arch that does not have a `SCRIPTS_<osclass>` definition
`SCRIPTS_IOC += <name>`
 Script <name> will be installed for IOC type archs.
`SCRIPTS_IOC_<osclass> += <name>`
 Script <name> will be installed for all IOC type archs of the specified osclass.
`SCRIPTS_IOC_DEFAULT += <name>`
 Script <name> will be installed for any IOC type arch that does not have a `SCRIPTS_IOC_<osclass>` definition
`SCRIPTS_HOST += <name>`
 Script <name> will be installed for HOST type archs.
`SCRIPTS_HOST_<osclass> += <name>`
 Script <name> will be installed for all HOST type archs of the specified osclass.
`SCRIPTS_HOST_DEFAULT += <name>`
 Script <name> will be installed for any HOST type arch that does not have a `SCRIPTS_HOST_<osclass>` definition

Definitions of the form:

```
SCRIPTS_<osclass> += <name1>
SCRIPTS_DEFAULT += <name2>
```

results in the <name1> script being installed from the src directory to the `$(INSTALL_LOCATION)/bin/<arch>` directories for all target archs of the specified os class <osclass> and the <name2> script installed into the `$(INSTALL_LOCATION)/bin/<arch>` directories of all other target archs.

4.6.19 Include files

A definition of the form:

```
INC += <name>.h
```

results in file <name>.h being installed or created and installed to the `$(INSTALL_LOCATION)/include` directory.

Definitions of the form:

```
INC_DEFAULT += <name>.h
INC_<osclass> += <name>.h
```

results in file <name>.h being installed or created and installed into the appropriate `$(INSTALL_LOCATION)/include/os/<osclass>` directory..

4.6.20 Html and Doc files

A definition of the form:

```
HTMLS_DIR = <dirname>
HTMLS += <name>
```

results in file <name> being installed from the src directory to the `$(INSTALL_LOCATION)/html/<dirname>` directory.

A definition of the form:

```
DOCS += <name>
```

results in file <name> being installed from the src directory to the `$(INSTALL_LOCATION)/doc` directory.

4.6.21 Templates

Adding definitions of the form

```
TEMPLATES_DIR = <dirname>  
TEMPLATES += <name>
```

results in the file <name> being installed from the src directory to the \$(INSTALL_LOCATION)/templates/<dirname> directory. If a directory structure of template files is to be installed, the template file names may include a directory prefix.

4.6.22 Lex and yacc

If a <name>.c source file specified in a Makefile definition is not found in the source directory, gnumake will try to build it from <name>.y and <name>_lex.l files in the source directory.

4.6.23 Products

A product executable is created for each <arch> and installed into \$(INSTALL_LOCATION)/bin/<arch> by specifying its name and the name of either the object or source files containing code for the product. An object or source file name can appear with or without a directory prefix. Object files should contain a directory prefix. If the file has a directory prefix e.g. \$(EPICS_BASE_BIN), the file is taken from the specified location. If a directory prefix is not present, make will look in the source directories for a file with the specified name or try build it using existing rules. An executable filename suffix appropriate for the target arch (e.g. .exe) may be appended to the filename when the file is created.

PROD specifications in the Makefile for vxWorks target architectures create a combined object file with library references resolved and a corresponding .munch file.

```
PROD_HOST += <name>  
<name>_SRC += <srcname>.c
```

results in the executable <name> being built for each HOST architecture, <arch>, from a <srcname>.c file. Then <name> is installed into the \$(INSTALL_LOCATION)/bin/<arch> directory.

4.6.23.1 Specifying the product name.

Any of the following can be specified:

```
PROD += <name>  
    Product <name> will be created for every target arch.  
PROD_<osclass> += <name>  
    Product <name> will be created for all archs of the specified osclass.  
PROD_DEFAULT += <name>  
    Product <name> will be created for any arch that does not have a PROD_<osclass> definition  
  
PROD_IOC += <name>  
    Product <name> will be created for IOC type archs.  
PROD_IOC_<osclass> += <name>  
    Product <name> will be created for all IOC type archs of the specified osclass.  
PROD_IOC_DEFAULT += <name>  
    Product <name> will be created for any IOC type arch that does not have a PROD_IOC_<osclass>  
    definition  
  
PROD_HOST += <name>  
    Product <name> will be created for HOST type archs.  
PROD_HOST_<osclass> += <name>
```

Product <name> will be created for all HOST type archs of the specified osclass.

PROD_HOST_DEFAULT += <name>

Product <name> will be created for any HOST type arch that does not have a PROD_HOST_<osclass> definition

4.6.23.2 Specifying product object file names

Object files which have filenames with a ".o" or ".obj" suffix are defined as follows and can be specified without the suffix but should have the directory prefix

USR_OBJS += <name>

Object files will be used in builds of all products and libraries

USR_OBJS_<osclass> += <name>

Object files will be used in builds of all products and libraries for archs with the specified osclass.

USR_OBJS_DEFAULT += <name>

Object files will be used in builds of all products and libraries for archs without a USR_OBJS_<osclass> definition specified.

PROD_OBJS += <name>

Object files will be used in builds of all products

PROD_OBJS_<osclass> += <name>

Object files will be used in builds of all products for archs with the specified osclass.

PROD_OBJS_DEFAULT += <name>

Object files will be used in builds of all products for archs without a PROD_OBJS_<osclass> definition specified.

<prodname>_OBJS += <name>

Object files will be used for all builds of the named product

<prodname>_OBJS_<osclass> += <name>

Object files will be used in builds of the named product for archs with the specified osclass.

<prodname>_OBJS_DEFAULT += <name>

Object files will be used in builds of the named product for archs without a <prodname>_OBJS_<osclass> definition specified.

Combined object files, from R3.13 built modules and applications which have file names that do not include a ".o" or ".obj" suffix (e.g. xyzLib) are defined as follows:

USR_OBJLIBS += <name>

Combined object files will be used in builds of all libraries and products.

USR_OBJLIBS_<osclass> += <name>

Combined object files will be used in builds of all libraries and products for archs of the specified osclass.

USR_OBJLIBS_DEFAULT += <name>

Combined object files will be used in builds of all libraries and products for archs without a USR_OBJLIBS_<osclass> definition specified.

PROD_OBJLIBS += <name>

Combined object files will be used in builds of all products.

PROD_OBJLIBS_<osclass> += <name>

Combined object files will be used in builds of all products for archs of the specified osclass.

PROD_OBJLIBS_DEFAULT += <name>

Combined object files will be used in builds of all products for archs without a PROD_OBJLIBS_<osclass> definition specified.

<prodname>_OBJLIBS += <name>

Combined object files will be used for all builds of the named product.

<prodname>_OBJLIBS_<osclass> += <name>

Combined object files will be used in builds of the named product for archs with the specified osclass.

`<prodname>_OBJLIBS_DEFAULT += <name>`

Combined object files will be used in builds of the named product for archs without a `<prodname>_OBJLIBS_<osclass>` definition specified.

`<prodname>_LDOBJ += <name>`

Object files will be used for all builds of the named product. (deprecated)

`<prodname>_LDOBJ_<osclass> += <name>`

Object files will be used in builds of the name product for archs with the specified `osclass`. (deprecated)

`<prodname>_LDOBJ_DEFAULT += <name>`

Object files will be used in builds of the product for archs without a `<prodname>_LDOBJ_<osclass>` definition specified. (deprecated)

4.6.23.3 Specifying product source file names

Source file names, which must have a suffix, are defined as follows:

`SRCS += <name>`

Source files will be used for all defined libraries and products.

`SRCS_<osclass> += <name>`

Source files will be used for all defined libraries and products for all archs of the specified `osclass`.

`SRCS_DEFAULT += <name>`

Source files will be used for all defined libraries and products for any arch that does not have a `SRCS_<osclass>` definition

`USR_SRCS += <name>`

Source files will be used for all products and libraries.

`USR_SRCS_<osclass> += <name>`

Source files will be used for all defined products and libraries for all archs of the specified `osclass`.

`USR_SRCS_DEFAULT += <name>`

Source files will be used for all defined products and libraries for any arch that does not have a `USR_SRCS_<osclass>` definition

`PROD_SRCS += <name>`

Source files will be used for all products.

`PROD_SRCS_<osclass> += <name>`

Source files will be used for all defined products for all archs of the specified `osclass`.

`PROD_SRCS_DEFAULT += <name>`

Source files will be used for all defined products for any arch that does not have a `PROD_SRCS_<osclass>` definition

`<prodname>_SRCS += <name>`

Source file will be used for the named product.

`<prodname>_SRCS_<osclass> += <name>`

Source files will be used for named product for all archs of the specified `osclass`.

`<prodname>_SRCS_DEFAULT += <name>`

Source files will be used for named product for any arch that does not have a `<prodname>_SRCS_<osclass>` definition

4.6.23.4 Specifying libraries to be linked when creating the product

For each library name specified which is not a system library nor a library from `EPICS_BASE`, a `<name>_DIR` definition must be present in the Makefile to specify the location of the library.

Library names, which must not have a directory and "lib" prefix nor a suffix, are defined as follows:

`PROD_LIBS += <name>`
Libraries to be used when linking all defined products.

`PROD_LIBS_<osclass> += <name>`
Libraries to be used or all archs of the specified osclass when linking all defined products.

`PROD_LIBS_DEFAULT += <name>`
Libraries to be used for any arch that does not have a `PROD_LIBS_<osclass>` definition when linking all defined products.

`USR_LIBS += <name>`
Libraries to be used when linking all defined products.

`USR_LIBS_<osclass> += <name>`
Libraries to be used or all archs of the specified osclass when linking all defined products.

`USR_LIBS_DEFAULT += <name>`
Libraries to be used for any arch that does not have a `USR_LIBS_<osclass>` definition when linking all defined products.

`<prodname>_LIBS += <name>`
Libraries to be used for linking the named product.

`<prodname>_LIBS_<osclass> += <name>`
Libraries will be used for all archs of the specified osclass for linking named product.

`<prodname>_LIBS_DEFAULT += <name>`
Libraries to be used for any arch that does not have a `<prodname>_LIBS_<osclass>` definition when linking named product.

`SYS_PROD_LIBS += <name>`
System libraries to be used when linking all defined products.

`SYS_PROD_LIBS_<osclass> += <name>`
System libraries to be used for all archs of the specified osclass when linking all defined products.

`SYS_PROD_LIBS_DEFAULT += <name>`
System libraries to be used for any arch that does not have a `PROD_LIBS_<osclass>` definition when linking all defined products.

`<prodname>_SYS_LIBS += <name>`
System libraries to be used for linking the named product.

`<prodname>_SYS_LIBS_<osclass> += <name>`
System libraries will be used for all archs of the specified osclass for linking named product.

`<prodname>_SYS_LIBS_DEFAULT += <name>`
System libraries to be used for any arch that does not have a `<prodname>_LIBS_<osclass>` definition when linking named product.

4.6.23.5 The order of dependant libraries

Dependant library names appear in the following order on a product link line:

1. `<prodname>_LIBS`
 2. `<prodname>_LIBS_<osclass>` or `<prodname>_LIBS_DEFAULT`
 3. `PROD_LIBS`
 4. `PROD_LIBS_<osclass>` or `PROD_LIBS_DEFAULT`
 5. `USR_LIBS`
 6. `USR_LIBS_<osclass>` or `USR_LIBS_DEFAULT`
 7. `<prodname>_SYS_LIBS`
 8. `<prodname>_SYS_LIBS_<osclass>` or `<prodname>_SYS_LIBS_DEFAULT`
-

9. PROD_SYS_LIBS
10. PROD_SYS_LIBS_<osclass> or PROD_SYS_LIBS_DEFAULT
11. USR_SYS_LIBS
12. USR_SYS_LIBS_<osclass> or USR_SYS_LIBS_DEFAULT

4.6.23.6 Specifying product version number

On WIN32 only a product version number can be specified as follows:

```
PROD_VERSION += <version>
```

This results in "/version:\$(PROD_VERSION)" link option.

4.6.23.7 Product static builds

Product executables can be linked with either archive versions or shared versions of EPICS libraries. Shared versions of system libraries will always be used in product linking. The definition of `STATIC_BUILD` (YES/NO) in `base/configure/CONFIG_SITE` determines which EPICS libraries to use. When `STATIC_BUILD` is NO, shared libraries will be used. (`SHARED_LIBRARIES` must be set to YES.) The default definition for `STATIC_BUILD` in the EPICS base `CONFIG_SITE` distribution file is NO. A `STATIC_BUILD` definition in a Makefile will override the definition in `CONFIG_SITE`.

4.6.24 Test Products

Test products are product executables that are created but not installed into `$(INSTALL_LOCATION)/bin/<arch>` directories. Test product libraries, source, and object files are specified in exactly the same way as regular products.

Any of the following can be specified:

```
TESTPROD += <name>
```

Test product <name> will be created for every target arch.

```
TESTPROD_<osclass> += <name>
```

Test product <name> will be created for all archs of the specified osclass.

```
TESTPROD_DEFAULT += <name>
```

Test product <name> will be created for any arch that does not have a `TESTPROD_<osclass>` definition

```
TESTPROD_IOC += <name>
```

Test product <name> will be created for IOC type archs.

```
TESTPROD_IOC_<osclass> += <name>
```

Test product <name> will be created for all IOC type archs of the specified osclass.

```
TESTPROD_IOC_DEFAULT += <name>
```

Test product <name> will be created for any IOC type arch that does not have a `TESTPROD_IOC_<osclass>` definition

```
TESTPROD_HOST += <name>
```

Test product <name> will be created for HOST type archs.

```
TESTPROD_HOST_<osclass> += <name>
```

Test product <name> will be created for all HOST type archs of the specified osclass.

```
TESTPROD_HOST_DEFAULT += <name>
```

Test product <name> will be created for any HOST type arch that does not have a `TESTPROD_HOST_<osclass>` definition

4.6.25 Test Scripts

Test scripts are perl scripts whose names end in `.t` that get executed to satisfy the `runtests` make target. They are run by the perl `Test::Harness` library, and should send output to `stdout` following the Test Anything Protocol. Any of the following can be specified, although only `TESTSCRIPTS_HOST` is currently useful:

```
TESTSCRIPTS += <name>
    Test script <name> will be created for every target arch.
TESTSCRIPTS_<osclass> += <name>
    Test script <name> will be created for all archs of the specified osclass.
TESTSCRIPTS_DEFAULT += <name>
    Test script <name> will be created for any arch that does not have a TESTSCRIPTS_<osclass> definition

TESTSCRIPTS_IOC += <name>
    Test script <name> will be created for IOC type archs.
TESTSCRIPTS_IOC_<osclass> += <name>
    Test script <name> will be created for all IOC type archs of the specified osclass.
TESTSCRIPTS_IOC_DEFAULT += <name>
    Test script <name> will be created for any IOC type arch that does not have a
    TESTSCRIPTS_IOC_<osclass> definition

TESTSCRIPTS_HOST += <name>
    Test script <name> will be created for HOST type archs.
TESTSCRIPTS_HOST_<osclass> += <name>
    Test script <name> will be created for all HOST type archs of the specified osclass.
TESTSCRIPTS_HOST_DEFAULT += <name>
    Test script <name> will be created for any HOST type arch that does not have a
    TESTSCRIPTS_HOST_<osclass> definition.
```

If a name in one of the above variables matches a regular executable program name (normally generated as a test product) with `".t"` appended, a suitable perl script will be generated that will execute that program directly; this makes it simple to run programs that use the `epicsUnitTest` routines in `libCom`. A test script written in Perl with a name ending `.plt` will be copied into the `O.<arch>` directory with the ending changed to `.t`; such scripts will usually use the perl `Test::Simple` or `Test::More` libraries.

4.6.26 Miscellaneous Targets

A definition of the form:

```
TARGETS += <name>
```

results in the file `<name>` being built in the `O.<arch>` directory from existing rules and files in the source directory. These target files are not installed.

4.6.27 Installing Other Binaries

Definitions of the form:

```
BIN_INSTALLS += <name>
BIN_INSTALLS += <dir>/<name>
BIN_INSTALLS_DEFAULT += <name>
BIN_INSTALLS_<osclass> += <name>
```

will result in the named files being installed to the appropriate $\$(INSTALL_LOCATION)/bin/<arch>$ directory. The file $<name>$ can appear with or without a directory prefix. If the file has a directory prefix e.g. $\$(EPICS_BASE_BIN)$, it is copied from the specified location. If a directory prefix is not present, make will look in the source directory for the file.

4.6.28 Installing Other Libraries

Definitions of the form:

```
LIB_INSTALLS += <name>
LIB_INSTALLS += <dir>/<name>
LIB_INSTALLS_DEFAULT += <name>
LIB_INSTALLS_<osclass> += <name>
```

result in files being installed to the appropriate $\$(INSTALL_LOCATION)/lib/<arch>$ directory. The file $<name>$ can appear with or without a directory prefix. If the file has a directory prefix e.g. $\$(EPICS_BASE_LIB)$, it is copied from the specified location. If a directory prefix is not present, make will look in the source directory for the file.

4.6.29 Win32 resource files

Definitions of the form:

```
RCS += <name>    Resource files for all products and libraries.
RCS_<osclass> += <name>

PROD_RCS += <name> Resource files for all products.
PROD_RCS_<osclass> += <name>
PROD_RCS_DEFAULT += <name>

LIB_RCS += <name> Resource files for all libraries.
LIB_RCS_<osclass> += <name>
LIB_RCS_DEFAULT += <name>

<name>_RCS += <name> Resource files for specified product or library.
<name>_RCS_<osclass> += <name>
<name>_RCS_DEFAULT += <name>
```

result in resource files (*.res files) being created from the specified *.rc files and linked into the prods and/or libraries.

4.6.30 TCL libraries

Definitions of the form:

```
TCLLIBNAME += <name>
TCLINDEX += <name>
```

result in the specified tcl files being installed to the $\$(INSTALL_LOCATION)/lib/<arch>$ directory.

4.6.31 Java class files

Java class files can be created by the javac tool into $\$(INSTALL_JAVA)$ or into the O.Common subdirectory, by specifying the name of the java class file in the Makefile. Command line options for the javac tool can be specified. The configuration files set the java c option "-sourcepath ..:../..".

Any of the following can be specified:

`JAVA += <name>.java`

The <name>.java file will be used to create the <name>.class file in the \$(INSTALL_JAVA) directory.

`TESTJAVA += <name>.java`

The <name>.java files will be used to create the <name>.class file in the O.Common subdirectory.

`USR_JAVACFLAGS += <name>`

The javac option <name> will be used on the javac command lines.

4.6.31.1 Example 1

In this example, three class files are created in \$(INSTALL_LOCATION)/javalib/mytest. The javac deprecation flag is used to list the description of each use or override of a deprecated member or class.

`JAVA = mytest/one.java`

`JAVA = mytest/two.java`

`JAVA = mytest/three.java`

`USR_JAVACFLAGS = -deprecation`

4.6.31.2 Example 2

In this example, the test.class file is created in the O.Common subdirectory.

`TESTJAVA = test.java`

4.6.32 Java jar file

A single java jar file can be created using the java jar tool and installed into \$(INSTALL_JAVA) (i.e. \$(INSTALL_LOCATION)/javalib) by specifying its name, and the names of its input files to be included in the created jar file. The jar input file names must appear with a directory prefix.

Any of the following can be specified:

`JAR += <name>`

The <name> jar file will be created and installed into the \$(INSTALL_JAVA) directory.

`JAR_INPUT += <name>`

Names of images, audio files and classes files to be included in the jar file.

`JAR_MANIFEST += <name>`

The preexisting manifest file will be used for the created jar file.

4.6.32.1 Example 1

In this example, all the class files created by the current Makefile's "JAVA+=" definitions, are placed into a file named mytest1.jar. A manifest file will be automatically generated for the jar.

Note: \$(INSTALL_CLASSES) is set to \$(addprefix \$(INSTALL_JAVA)/,\$(CLASSES)) in the EPICS base configure files.

`JAR = mytest1.jar`

`JAR_INPUT = $(INSTALL_CLASSES)`

4.6.32.2 Example 2

In this example, three class files are created and placed into a new jar archive file named mytest2.jar. An existing manifest file, mytest2.mf is put into the new jar file.

`JAR = mytest2.jar`

```
JAR_INPUT = $(INSTALL_JAVA)/mytest/one.class
JAR_INPUT = $(INSTALL_JAVA)/mytest/two.class
JAR_INPUT = $(INSTALL_JAVA)/mytest/three.class
JAR_MANIFEST = mytest2.mf
```

4.6.33 Java native method C header files

A C header files for use with java native methods will be created by the javah tool in the O.Common subdirectory by specifying the name of the header file to be created. The name of the java class file used to generate the header is derived from the name of the header file. Underscores (_) are used as a header file name delimiter. Command line options for the javah tool can be specified.

Any of the following can be specified:

```
JAVAINC += <name>.h
    The <name>.h header file will be created in the O.Common subdirectory.
USR_JAVAHFLAGS += <name>
    The javah option <name> will be used on the javah tool command line.
```

4.6.33.1 Example

In this example, the C header `xx_yy_zz.h` will be created in the `$(COMMON_DIR)` subdirectory from the class `xx.yy.zz` (i.e. the java class file `$(INSTALL_JAVA)/xx/yy/zz.class`). The option `"-old"` will tell javah to create old JDK1.0 style header files.

```
JAVAINC = xx_yy_zz.h
USR_JAVAHFLAGS = -old
```

4.7 Table of Makefile definitions

Definitions given below containing `<osclass>` are used when building for target archs of a specific `osclass`, and the `<osclass>` part of the name should be replaced by the desired `osclass`, e.g. `solaris`, `vxWorks`, etc. If a `_DEFAULT` setting is given but a particular `<osclass>` requires that the default not apply and there are no items in the definition that apply for that `<osclass>`, the value `"-nil"` should be specified in the relevant Makefile definition.

Build Option	Description
Products to be built (host type archs only)	
PROD	products (names without execution suffix) to build and install. Specify xyz to build executable xyz on Unix and xyz.exe on WIN32
PROD_<osclass>	os class specific products to build and install for <osclass> archs only
PROD_DEFAULT	products to build and install for archs with no PROD_<osclass> specified
PROD_IOC	products to build and install for ioc type archs
PROD_IOC_<osclass>	os specific products to build and install for ioc type archs
PROD_IOC_DEFAULT	products to build and install for ioc type arch systems with no PROD_IOC_<osclass> specified

Build Option	Description
PROD_HOST	products to build and install for host type archs.
PROD_HOST_<osclass>	os class specific products to build and install for <osclass> type archs
PROD_HOST_DEFAULT	products to build and install for arch with no PROD_HOST_<osclass> specified
Test products to be built	
TESTPROD	test products (names without execution suffix) to build but not install
TESTPROD_<osclass>	os class specific test products to build but not install
TESTPROD_DEFAULT	test products to build but not install for archs with no TESTPROD_<osclass> specified
TESTPROD_IOC	test products to build and install for ioc type archs
TESTPROD_IOC_<osclass>	os specific test products to build and install for ioc type archs
TESTPROD_IOC_DEFAULT	test products to build and install for ioc type arch systems with no TESTPROD_IOC_<osclass> specified
TESTPROD_HOST	testproducts to build and install for host type archs.
TESTPROD_HOST_<osclass>	os class specific testproducts to build and install for <osclass> type archs
TESTPROD_HOST_DEFAULT	test products to build and install for arch with no TESTPROD_HOST_<osclass> specified
TESTSCRIPTS	test scripts (names with .t suffix) to build but not install
TESTSCRIPTS_<osclass>	os class specific test scripts to build but not install
TESTSCRIPTS_DEFAULT	test scripts to build but not install for archs with no TESTSCRIPTS_<osclass> specified
TESTSCRIPTS_IOC	test scripts to build and install for ioc type archs
TESTSCRIPTS_IOC_<osclass>	os specific test scripts to build and install for ioc type archs
TESTSCRIPTS_IOC_DEFAULT	test scripts to build and install for ioc type arch systems with no TESTSCRIPTS_IOC_<osclass> specified
TESTSCRIPTS_HOST	test scripts to build and install for host type archs.
TESTSCRIPTS_HOST_<osclass>	os class specific testscripts to build and install for <osclass> type archs
TESTSCRIPTS_HOST_DEFAULT	test scripts to build and install for arch with no TESTSCRIPTS_HOST_<osclass> specified
Libraries to be built	
LIBRARY	name of library to build and install. The name should NOT include a prefix or extension e.g. specify Ca to build libCa.a on Unix, Ca.lib, CaObj.lib, or Ca.dll on WIN32
LIBRARY_<osclass>	os specific libraries to build and install

Chapter 4: EPICS Build Facility

Table of Makefile definitions

Build Option	Description
LIBRARY_DEFAULT	libraries to build and install for archs with no LIBRARY_<osclass> specified
LIBRARY_IOC	name of library to build and install for ioc type archs. The name should NOT include a prefix or extension e.g. specify Ca to build libCa.a on Unix, Ca.lib, CaObj.lib, or Ca.dll on WIN32
LIBRARY_IOC_<osclass>	os specific libraries to build and install for ioc type archs
LIBRARY_IOC_DEFAULT	libraries to build and install for ioc type arch systems with no LIBRARY_IOC_<osclass> specified
LIBRARY_HOST	name of library to build and install for host type archs. The name should NOT include a prefix or extension, e.g. specify Ca to build libCa.a on Unix, Ca.lib, CaObj.lib, or Ca.dll on WIN32
LIBRARY_HOST_<osclass>	os class specific libraries to build and install for host type archs
LIBRARY_HOST_DEFAULT	libraries to build and install for host type arch systems with no LIBRARY_HOST_<osclass> specified
SHARED_LIBRARIES	build shared libraries? Must be YES or NO
SHRLIB_VERSION	shared library version number
Loadable libraries to be built	
LOADABLE_LIBRARY	name of loadable library to build and install. The name should NOT include a prefix or extension e.g. specify Ca to build libCa.so on Unix and Ca.dll on WIN32
LOADABLE_LIBRARY_<osclass>	os specific loadable libraries to build and install
LOADABLE_LIBRARY_DEFAULT	loadable libraries to build and install for archs with no LOADABLE_LIBRARY_<osclass> specified
LOADABLE_LIBRARY_HOST	name of loadable library to build and install for host type archs. The name should NOT include a prefix or extension, e.g. specify test to build libtest.so on Unix and test.dll on WIN32
LOADABLE_LIBRARY_HOST_<osclass>	os class specific loadable libraries to build and install for host type archs
LOADABLE_LIBRARY_HOST_DEFAULT	loadable libraries to build and install for host type arch systems with no LOADABLE_LIBRARY_HOST_<osclass> specified
Combined object files (vxWorks only)	
OBJLIB	name of a combined object file library and corresponding munch file to build and install. The name will have a Library suffix appended
OBJLIB_vxWorks	same as OBJLIB
OBJLIB_SRCS	source files to build the OBJLIB
OBJLIB_OBJS	object files to include in OBJLIB
Product and library source files	
SRCS	source files to build all PRODs and LIBRARYs

Build Option	Description
SRCS_<osclass>	osclass specific source files to build all PRODs and LIBRARYs
SRCS_DEFAULT	source file to build all PRODs and LIBRARYs for archs with no SRCS_<osclass> specified
USR_SRCS	source files to build all PRODs and LIBRARYs
USR_SRCS_<osclass>	osclass specific source files to build all PRODs and LIBRARYs
USR_SRCS_DEFAULT	source file to build all PRODs and LIBRARYs for archs with no SRCS_<osclass> specified
PROD_SRCS	source files to build all PRODs
PROD_SRCS_<osclass>	osclass specific source files to build all PRODs
PROD_SRCS_DEFAULT	source files needed to build PRODs for archs with no SRCS_<osclass> specified
LIB_SRCS	source files for building LIBRARY (e.g. LIB_SRCS=la.c lb.c lc.c)
LIB_SRCS_<osclass>	os-specific library source files
LIB_SRCS_DEFAULT	library source files for archs with no LIB_SRCS_<osclass> specified
LIBSRCS	source files for building LIBRARY (deprecated)
LIBSRCS_<osclass>	os-specific library source files (deprecated)
LIBSRCS_DEFAULT	library source files for archs with no LIBSRCS_<osclass> specified (deprecated)
<name>_SRCS	source files to build a specific PROD or LIBRARY
<name>_SRCS_<osclass>	os specific source files to build a specific PROD or LIBRARY
<name>_SRCS_DEFAULT	source files needed to build a specific PROD or LIBRARY for archs with no <prod>_SRCS_<osclass> specified
Product and library object files	
USR_OBJs	object files, specified without suffix, to build all PRODs and LIBRARYs
USR_OBJs_<osclass>	osclass specific object files, specified without suffix, to build all PRODs and LIBRARYs
USR_OBJs_DEFAULT	object files, specified without suffix, needed to build PRODs and LIBRARYs for archs with no OBJs_<osclass> specified
PROD_OBJs	object files, specified without suffix, to build all PRODs
PROD_OBJs_<osclass>	osclass specific object files, specified without suffix, to build all PRODs
PROD_OBJs_DEFAULT	object files, specified without suffix, needed to build PRODs for archs with no OBJs_<osclass> specified
LIB_OBJs	object files, specified without suffix, for building all LIBRARYs (e.g. LIB_OBJs+=\$(AB_BIN)/la \$(AB_BIN)/lb)
LIB_OBJs_<osclass>	os-specific library object files, specify without suffix,

Chapter 4: EPICS Build Facility

Table of Makefile definitions

Build Option	Description
LIB_OBJS_DEFAULT	library object files, specified without suffix, for archs with no LIB_OBJS_<osclass> specified
<name>_OBJS	object files, specified without suffix, to build a specific PROD or LIBRARY
<name>_OBJS_<osclass>	os specific object files, specified without suffix, to build a specific PROD or LIBRARY
<name>_OBJS_DEFAULT	object files, without suffix, needed to build a specific PROD or LIBRARY for archs with no <prod>_OBJS_<osclass> specified
Product and library R3.13 combined object files	
USR_OBJLIBS	combined object files with filenames that do not have a suffix, needed for building all PRODs and LIBRARYs (e.g. USR_OBJLIBS+=\$(XYZ_BIN)/xyzLib)
USR_OBJLIBS_<osclass>	os-specific combined object files with filenames that do not have a suffix for building all PRODs and LIBRARYs
USR_OBJLIBS_DEFAULT	combined object files with filenames that do not have a suffix, for archs with no USR_OBJLIBS_<osclass> specified for building all PRODs and LIBRARYs
PROD_OBJLIBS	combined object files with filenames that do not have a suffix, needed for building all PRODs (e.g. PROD_OBJLIBS+=\$(XYZ_BIN)/xyzLib)
PROD_OBJLIBS_<osclass>	os-specific combined object files with filenames that do not have a suffix for building all PRODs
PROD_OBJLIBS_DEFAULT	combined object files with filenames that do not have a suffix, for archs with no PROD_OBJLIBS_<osclass> specified for building all PRODs
LIB_OBJLIBS	combined object files with filenames that do not have a suffix, needed for building all LIBRARYs (e.g. LIB_OBJLIBS+=\$(XYZ_BIN)/xyzLib)
LIB_OBJLIBS_<osclass>	os-specific combined object files with filenames that do not have a suffix for building all LIBRARYs
LIB_OBJLIBS_DEFAULT	combined object files with filenames that do not have a suffix, for archs with no LIB_OBJLIBS_<osclass> specified for building all LIBRARYs
<name>_OBJLIBS	combined object files with filenames that do not have a suffix, needed to build a specific PROD or LIBRARY
<name>_OBJLIBS_<osclass>	os specific combined object files with filenames that do not have a suffix, to build a specific PROD or LIBRARY
<name>_OBJLIBS_DEFAULT	combined object files with filenames that do not have a suffix, needed to build a specific PROD or LIBRARY for archs with no <name>_OBJLIBS_<osclass> specified
<name>_LDOBJS	combined object files with filenames that do not have a suffix, needed to build a specific PROD or LIBRARY (deprecated)

Build Option	Description
<name>_LDOBJS_<osclass>	os specific combined object files with filenames that do not have a suffix, to build a specific PROD or LIBRARY (deprecated)
<name>_LDOBJS_DEFAULT	combined object files with filenames that do not have a suffix, needed to build a specific PROD or LIBRARY for archs with no <name>_LDOBJS_<osclass> specified (deprecated)
Product and library dependant libraries	
<name>_DIR	directory to search for the specified lib. (For libs listed in all PROD_LIBS, LIB_LIBS, <name>_LIBS and USR_LIBS listed below)
USR_LIBS	load libraries (e.g. Xt X11) for all products and libraries
USR_LIBS_<osclass>	os specific load libraries for all makefile links
USR_LIBS_DEFAULT	load libraries for systems with no USR_LIBS_<osclass> specified libs
<name>_LIBS	named prod or library specific ld libraries (e.g. probe_LIBS=X11 Xt)
<name>_LIBS_<osclass>	os-specific libs needed to link named prod or library
<name>_LIBS_DEFAULT	libs needed to link named prod or library for systems with no <name>_LIBS_<osclass> specified
PROD_LIBS	libs needed to link every PROD
PROD_LIBS_<osclass>	os-specific libs needed to link every PROD
PROD_LIBS_DEFAULT	libs needed to link every PROD for archs with no PROD_LIBS_<osclass> specified
LIB_LIBS	libraries to be linked with every library being created
LIB_LIBS_<osclass>	os class specific libraries to be linked with every library being created
LIB_LIBS_DEFAULT	libraries to be linked with every library being created for archs with no LIB_LIBS_<osclass> specified
USR_SYS_LIBS	system libraries (e.g. Xt X11) for all products and libraries
USR_SYS_LIBS_<osclass>	os class specific system libraries for all makefile links
USR_SYS_LIBS_DEFAULT	system libraries for archs with no USR_SYS_LIBS_<osclass> specified
<name>_SYS_LIBS	named prod or library specific system ld libraries
<name>_SYS_LIBS_<osclass>	os class specific system libs needed to link named prod or library
<name>_SYS_LIBS_DEFAULT	system libs needed to link named prod or library for systems with no <name>_SYS_LIBS_<osclass> specified
PROD_SYS_LIBS	system libs needed to link every PROD
PROD_SYS_LIBS_<osclass>	os class specific system libs needed to link every PROD
PROD_SYS_LIBS_DEFAULT	system libs needed to link every PROD for archs with no PROD_SYS_LIBS_<osclass> specified
LIB_SYS_LIBS	system libraries to be linked with every library being created

Chapter 4: EPICS Build Facility

Table of Makefile definitions

Build Option	Description
LIB_SYS_LIBS_<osclass>	os class specific system libraries to be linked with every library being created
LIB_SYS_LIBS_DEFAULT	system libraries to be linked with every library being created for archs with no LIB_SYS_LIBS_<osclass> specified
SYS_PROD_LIBS	system libs needed to link every PROD for all systems (deprecated)
SYS_PROD_LIBS_<osclass>	os class specific system libs needed to link every PROD (deprecated)
SYS_PROD_LIBS_DEFAULT	system libs needed to link every PROD for systems with no SYS_PROD_LIBS_<osclass> specified (deprecated)
Compiler flags	
USR_CFLAGS	C compiler flags for all systems
USR_CFLAGS_<T_A>	target architecture specific C compiler flags
USR_CFLAGS_<osclass>	os class specific C compiler flags
USR_CFLAGS_DEFAULT	C compiler flags for archs with no USR_CFLAGS_<osclass> specified
<name>_CFLAGS	file specific C compiler flags (e.g. xxxRecord_CFLAGS=-g)
<name>_CFLAGS_<T_A>	file specific C compiler flags for a specific target architecture
<name>_CFLAGS_<osclass>	file specific C compiler flags for a specific os class
USR_CXXFLAGS	C++ compiler flags for all systems (e.g. xyxMain_CXXFLAGS=-DSDDS)
USR_CXXFLAGS_<T_A>	target architecture specific C++ compiler flags
USR_CXXFLAGS_<osclass>	os-specific C++ compiler flags
USR_CXXFLAGS_DEFAULT	C++ compiler flags for systems with no USR_CXXFLAGS_<osclass> specified
<name>_CXXFLAGS	file specific C++ compiler flags
<name>_CXXFLAGS_<T_A>	file specific C++ compiler flags for a specific target architecture
<name>_CXXFLAGS_<osclass>	file specific C++ compiler flags for a specific osclass
USR_CPPFLAGS	C pre-processor flags (for all makefile compiles)
USR_CPPFLAGS_<T_A>	target architecture specific cpp flags
USR_CPPFLAGS_<osclass>	os specific cpp flags
USR_CPPFLAGS_DEFAULT	cpp flags for systems with no USR_CPPFLAGS_<osclass> specified
<name>_CPPFLAGS	file specific C pre-processor flags (e.g. xxxRecord_CPPFLAGS=-DDEBUG)
<name>_CPPFLAGS_<T_A>	file specific cpp flags for a specific target architecture
<name>_CPPFLAGS_<osclass>	file specific cpp flags for a specific os class
USR_INCLUDES	directories, with -I prefix, to search for include files (e.g. -I\$(EPICS_EXTENSIONS_INCLUDE))

Build Option	Description
USR_INCLUDES_<osclass>	directories, with -I prefix, to search for include files for a specific os class
USR_INCLUDES_DEFAULT	directories, with -I prefix, to search for include files for systems with no <name>_INCLUDES_<osclass> specified
<name>_INCLUDES	directories, with -I prefix, to search for include files when building a specific object file (e.g. -I\$(MOTIF_INC))
<name>_INCLUDES_<T_A>	file specific directories, with -I prefix, to search for include files for a specific target architecture
<name>_INCLUDES_<osclass>	file specific directories, with -I prefix, to search for include files for a specific os class
HOST_WARN	Are compiler warning messages desired for host type builds? (YES or NO) (default is YES)
CROSS_WARN	C cross-compiler warning messages desired (YES or NO) (default YES)
HOST_OPT	Is host build compiler optimization desired (default is NO optimization)
CROSS_OPT	Is cross-compiler optimization desired (YES or NO) (default is NO optimization)
CMPLR	C compiler selection, TRAD, ANSI or STRICT (default is STRICT)
CXXCMPLR	C++ compiler selection, NORMAL or STRICT (default is STRICT)
Linker options	
USR_LDFLAGS	linker options (for all makefile links)
USR_LDFLAGS_<osclass>	os specific linker options (for all makefile links)
USR_LDFLAGS_DEFAULT	linker options for systems with no USR_LDFLAGS_<osclass> specified
PROD_LDFLAGS	prod linker options
PROD_LDFLAGS_<osclass>	os specific prod linker options
PROD_LDFLAGS_DEFAULT	prod linker options for systems with no PROD_LDFLAGS_<osclass> specified
LIB_LDFLAGS	library linker options
LIB_LDFLAGS_<osclass>	os specific library linker options
LIB_LDFLAGS_DEFAULT	library linker options for systems with no LIB_LDFLAGS_<osclass> specified
<name>_LDFLAGS	prod or library specific linker options
<name>_LDFLAGS_<osclass>	prod or library specific linker flags for a specific os class
<name>_LDFLAGS_DEFAULT	linker options for systems with no <name>_LDFLAGS_<osclass> specified
STATIC_BUILD	Is static build desired (YES or NO) (default is NO). On win32 if STATIC_BUILD=YES then set SHARED_LIBRARIES=NO

Chapter 4: EPICS Build Facility

Table of Makefile definitions

Build Option	Description
Header files to be installed	
INC	list of include files to install into \$(INSTALL_DIR)/include
INC_<osclass>	os specific includes to installed under \$(INSTALL_DIR)/include/os/<osclass>
INC_DEFAULT	include files to install where no INC_<osclass> is specified
Perl, csh, tcl etc. script installation	
SCRIPTS	scripts to install for all systems
SCRIPTS_<osclass>	os-specific scripts to install
SCRIPTS_DEFAULT	scripts to install for systems with no SCRIPTS_<osclass> specified
SCRIPTS_IOC	scripts to install for ioc type archs.
SCRIPTS_IOC_<osclass>	os specific scripts to install for ioc type archs
SCRIPTS_IOC_DEFAULT	scripts to install for ioc type arch systems with no SCRIPTS_IOC_<osclass> specified
SCRIPTS_HOST	scripts to install for host type archs. T
SCRIPTS_HOST_<osclass>	os class specific scripts to install for host type archs
SCRIPTS_HOST_DEFAULT	scripts to install for host type arch systems with no OBJS_HOST_<osclass> specified
TCLLIBNAME	list of tcl scripts to install into \$(INSTALL_DIR)/lib/<osclass> (Unix hosts only)
TCLINDEX	name of tcl index file to create from TCLLIBNAME scripts
Object files	
The names in the following OBJS definitions should NOT include a suffix (.o or.obj).	
OBJS	object files to build and install for all system.
OBJS_<osclass>	os-specific object files to build and install.
OBJS_DEFAULT	object files to build and install for systems with no OBJS_<osclass> specified.
OBJS_IOC	object files to build and install for ioc type archs.
OBJS_IOC_<osclass>	os specific object files to build and install for ioc type archs
OBJS_IOC_DEFAULT	object files to build and install for ioc type arch systems with no OBJS_IOC_<osclass> specified
OBJS_HOST	object files to build and install for host type archs. T
OBJS_HOST_<osclass>	os class specific object files to build and install for host type archs
OBJS_HOST_DEFAULT	object files to build and install for host type arch systems with no OBJS_HOST_<osclass> specified
Documentation	

Build Option	Description
DOCS	text files to be installed into the \$(INSTALL_DIR)/doc directory
HTMLS_DIR	name install Hypertext directory name i.e. \$(INSTALL_DIR)/html/ \$(HTMLS_DIR)
HTMLS	hypertext files to be installed into the \$(INSTALL_DIR)/html/ \$(HTMLS_DIR) directory
TEMPLATES_DIR	template directory to be created as \$(INSTALL_DIR)/templates/ \$(TEMPLATE_DIR)
TEMPLATES	template files to be installed into \$(TEMPLATE_DIR)
Database Definition files	
DBD	database definition files to be installed or created and installed into \$(INSTALL_DBD).
DBDINC	names, without suffix, of menus or record database definitions and headers to be installed or created and installed.
USR_DBDFLAGS	optional flags for dbExpand. Currently only include path (-I <path>) and macro substitution (-S <substitution>) are supported.
DBD_INSTALLS	files from specified directory to install into \$(INSTALL_DBD) (e.g. DBD_INSTALLS = \$(APPNAME)/dbd/test.dbd
Database Files	
DB	database files to be installed or created and installed into \$(INSTALL_DB).
DB_INSTALLS	files from specified directory to install into \$(INSTALL_DB) (e.g. DB_INSTALLS = \$(APPNAME)/db/test.db
Options for other programs	
YACCOPT	yacc options
LEXOPT	lex options
SNCFLAGS	state notation language, snc, options
<name>_SNCFLAGS	product specific state notation language options
E2DB_FLAGS	e2db options
SCH2EDIF_FLAGS	sch2edif options
RANLIBFLAGS	ranlib options
Facilities for building Java programs	
CLASSES	names of Java class files to be built and installed
TESTCLASSES	names of Java class files to be built
JAVAINC	names of C header file to be created in O.Common subdirectory
JAR	name of Jar file to be built

Chapter 4: EPICS Build Facility

Table of Makefile definitions

Build Option	Description
JAR_INPUT	names of files to be included in JAR
JAR_MANIFEST	name of manifest file for JAR
USR_JAVACFLAGS	javac tool options
USR_JAVAHFLAGS	javah tool options
Facilities for Windows 95/NT resource (.rc) files	
RCS	resource files (<name>.rc) needed to build every PROD and LIBRARY
RCS_<osclass>	resource files (<name>.rc) needed to build every PROD and LIBRARY for ioc type archs
RCS_DEFAULT	resource files needed to build every PROD and LIBRARY for ioc type arch systems with no RCS_<osclass> specified
<name>_RCS	resource files needed to build a specific PROD or LIBRARY
<name>_RCS_<osclass>	os specific resource files to build a specific PROD or LIBRARY
<name>_RCS_DEFAULT	resource files needed to build a specific PROD or LIBRARY for ioc type arch systems with no RCS_<osclass> specified
Other definitions:	
USR_VPATH	list of directories
BIN_INSTALLS	files from specified directories to be installed into \$(INSTALL_BIN) (e.g. BIN_INSTALLS = \$(EPICS_BASE_BIN)/aiRecord\$(OBJ))
BIN_INSTALLS_<osclass>	os class specific files from specified directories to be installed into \$(INSTALL_BIN)
BIN_INSTALLS_DEFAULT	files from specified directories to be installed into \$(INSTALL_BIN) for target archs with no BIN_INSTALLS_<osclass> specified
LIB_INSTALLS	files from specified directories to be installed into \$(INSTALL_LIB)
LIB_INSTALLS_<osclass>	os class specific files from specified directories to be installed into \$(INSTALL_LIB)
LIB_INSTALLS_DEFAULT	files from specified directories to be installed into \$(INSTALL_LIB) for target archs with no LIB_INSTALLS_<osclass> specified
TARGETS	files to create but not install
INSTALL_LOCATION	installation directory (defaults to \$(TOP))

4.8 Configuration Files

4.8.1 Base Configure Directory

The base/configure directory has the following directory structure:

```
base/  
  configure/  
    os/  
    tools/
```

4.8.2 Base Configure File Descriptions

The configure files contain definitions and make rules to be included in the various makefiles.

CONFIG.CrossCommon

Definitions for all hosts and all targets for a cross build (host different than target).

CONFIG.gnuCommon

Definitions for all hosts and all targets for builds using the gnu compiler.

CONFIG_ADDONS

Definitions which setup the variables that have <osclass> and DEFAULT options.

CONFIG_BASE

EPICS base specific definitions.

CONFIG_BASE_VERSION

Definitions for the version number of EPICS base. This file is used for creating epicsVersion.h which is installed into base/include.

CONFIG_COMMON

Definitions common to all builds.

CONFIG_ENV

Default definitions of the EPICS environment variables. This file is used for creating envData.c which is included in the Com library.

CONFIG_SITE

File in which you add to or modify make variables in EPICS base. A definition normally overridden is:

```
CROSS_COMPILER_TARGET_ARCHS =
```

CONFIG_SITE_ENV

Defaults for site specific definitions of EPICS environment variables. This file is used for creating envData.c which is included in the Com library.

CONFIG

Include statements for all the other configure files. You can override any definitions in other CONFIG* files by placing override definitions at the end of this file.

RELEASE

Specifies the location of external products such as Tornado II and external <tops> such as EPICS base.

RULES

This file just includes the appropriate rules configuration file.

RULES.Db

Rules for building and installing database and database definition files. Databases generated from templates and/or CapFast schematics are supported.

RULES_ARCHS

Definitions and rules which allow building the make target for each target architecture.

RULES_BUILD

Build rules for the Makefiles

RULES_DIRS

Definitions and rules which allow building the make targets in each subdirectory. This file is included by Makefiles in directories with subdirectories to be built.

RULES_JAVA

Definitions and rules which allow building java class files and java jar files.

RULES_TOP

Rules specific to a <top> level directory e.g. uninstall and tar. It also includes the RULES_DIRS file.

RULES.ioc

Rules which allow building in the iocBoot/<iocname> directory of an ioc application.

Makefile

Definitions to allow creation of CONFIG_APP_INCLUDE and installation of the CONFIG* files into the \$(INSTALL_LOCATION) directory.

4.8.3 Base configure/os File Descriptions

The configure/os directory contains os specific make definitions. The naming convention for the files in this directory is CONFIG.<host>.<target> where <host> is either the arch for a specific host system or Common for all supported host systems and <target> is either the arch for a specific target system or Common for all supported target systems.

For example, the file CONFIG.Common.vxWorks-pentium will contain make definitions to be used for builds on all host systems when building for a vxWorks-pentium target system.

Also, if a group of host or target files have the same make definitions these common definitions can be moved to a new file which is then included in each host or target file. An example of this is all Unix hosts which have common definitions in a CONFIG.UnixCommon.Common file and all vxWorks targets with definitions in CONFIG.Common.vxWorksCommon.

The base/configure/os directory contains the following os-arch specific definitions

CONFIG.<host>.<target>

Specific host-target build definitions

CONFIG.Common.<target>

Specific target definitions for all hosts

CONFIG.<host>.Common

Specific host definitions for all targets

CONFIG.UnixCommon.Common

Definitions for Unix hosts and all target

CONFIG.<host>.vxWorksCommon

Specific host definitions for all vx targets

CONFIG_COMPAT

R3.13 arch compatibility definitions

CONFIG_SITE.<host>.<target>

Site specific host-target definitions

CONFIG_SITE.Common.<target>

Site specific target defs for all hosts

CONFIG_SITE.<host>.Common

Site specific host defs for all targets

4.8.4 Base configure/tool File Descriptions

The configure/tools directory contains Perl script tools used for the build. The tools currently in this directory are:

convertRelease.pl

This Perl script does consistency checks for the external <top> definitions in the RELEASE file and generates include directory, bin directory, and library directory definitions for these external <top>s. These definitions are

included into the CONFIG file for use by the application Makefiles. This script also creates -include statements for RULES_BUILD files from the external <top> definitions in the RELEASE file.

cp.pl

This Perl script copies an existing file.

cvsclean.pl

This perl script traverses a directory structure removing all .* cvs files.

filterWarnings.pl

This is a perl script that filters compiler warning output (for HP-UX).

fullpathname.pl

This perl script returns the fullpathname of a file.

installEpics.pl

This is a Perl script that installs build created files into the install directories.

makeIncludeDbd.pl

This perl script creates an include dbd file from file names

makeMakefile.pl

This is a perl script that creates a Makefile in the created O.<arch> directories.

makeMakefileInclude.pl

This perl script creates a file to be included by Makefiles. This file contains a build target's specific definitions and dependencies.

mkdir.pl

This perl script creates a directory like the Unix mkdir command.

mkmf.pl

This perl script generates include file dependencies for targets from source file include statements.

munch.pl

This is a perl script that creates a ctdt.c file for vxWorks target arch builds which lists the c++ static constructors and destructors. See munching in the vxWorks documentation for more information.

mv.pl

This perl script renames an existing file.

replaceVAR.pl

This is a perl script that changes VAR(xxx) style macros in CapFast generated databases into the \$(xxx) notation used in EPICS databases.

rm.pl

This perl script quietly removes an existing file.

4.9 Build Documentation Files

4.9.1 Base Documentation Directory

The base/documentation directory contains README files to help users setup and build epics/base.

4.9.2 Base Documentation File Descriptions

The files currently in the base/documentation directory are:

README.1st

Instructions for setup and building epics base

README.html

html version of README.1st

README.WIN32

Microsoft WIN32 specific instructions

README.niCpu030

NI cpu030 specific instructions

README.hpux

HPUX 11 (hpux-parisc) specific instructions

README.darwin.html

Installation notes for Mac OS X (Darwin)

BuildingR3.13AppsWithR3.14.html

Describes how to modify a R3.13 vxWorks application so that it builds with release R3.14.1.

ConvertingR3.13AppstoR3.14.html

Describes how to convert a R3.13 vxWorks application so that it contains a R3.14 configure directory and R3.14 Makefiles and builds with R3.14.1.

ConvertingR3.14.0alpha2AppstoBeta1.html

Describes how to modify a R3.14.0alpha1 application so that it builds with release R3.14.0beta1.

ConvertingR3.14.0beta1AppstoBeta2.html

Describes how to modify a R3.14.0beta1 application so that it builds with release R3.14.0beta2.

ConvertingR3.14.0beta2AppstoR3.14.1.html

Describes how to modify a R3.14.0beta2 application so that it builds with release R3.14.1.

ConvertingR3.14.*AppstoR3.14.*.html

Describes how to modify a R3.14.* application so that it builds with next release after R3.14.*.

BuildingR3.13ExtensionsWithR3.14.html

Describes how to modify a R3.13 extension so that it builds with release R3.14.1.

RELEASE_NOTES.html

Describes changes in the R3.14.1 release

KnownProblems.html

List of known problems in EPICS base R3.14.1.

ReleaseChecklist.html

Checklist of things that must be done when creating a new release of EPICS Base.

TSdriver.html

Synchronized Time Stamp Support documentation

4.10 Startup Files

4.10.1 Base Startup Directory

The base/startup directory contains scripts to help users set the required environment variables and path. The appropriate startup files should be executed before any EPICS builds.

4.10.2 Base Startup File Descriptions

The scripts currently in the base/startup directory are:

EpicsHostArch

c shell script to set EPICS_HOST_ARCH environment variable

EpicsHostArch.pl

perl script to set EPICS_HOST_ARCH environment variable

Site.profile

Unix bourne shell script to set path and environment variables

Site.cshrc

Unix c shell script to set path and environment variables

borland.bat

WIN32 bat file to set path and environment variables for building with Borland C++ compiler

win32.bat

WIN32 bat file to set path and environment variables for building with MS Visual C++ compilers

Chapter 5: Database Locking, Scanning, And Processing

5.1 Overview

Before describing particular components of the IOC software, it is helpful to give an overview of three closely related topics: Database locking, scanning, and processing. Locking is done to prevent two different tasks from simultaneously modifying related database records. Database scanning is the mechanism for deciding when records should be processed. The basics of record processing involves obtaining the current value of input fields and outputting the current value of output fields. As records become more complex so does the record processing.

One powerful feature of the DATABASE is that records can contain links to other records. This feature also causes considerable complication. Thus, before discussing locking, scanning, and processing, record links are described.

5.2 Record Links

A database record may contain links to other records. Each link is one of the following types:

- **INLINK**
OUTLINK
INLINKs and OUTLINKs can be one of the following:
 - constant link
Not discussed in this chapter
 - database link
A link to another record in the same IOC.
 - channel access link
A link to a record in another IOC. It is accessed via a special IOC client task. It is also possible to force a link to be a channel access link even it references a record in the same IOC.
 - hardware link
Not discussed in this chapter
- **FWDLINK**
A forward link refers to a record that should be processed whenever the record containing the forward link is processed. The following types are supported:
 - constant link
Ignored.
 - database link
A link to another record in the same IOC.
 - channel access link
A link to a record in another IOC or a link forced to be a channel access link. Unless the link references the PROC field it is ignored. If it does reference the PROC field a channel access put with a value of 1 is issued.

Links are defined in file `link.h`.

NOTE: This chapter discusses mainly database links.

5.3 Database Links

Database links are referenced by calling one of the following routines:

- **dbGetLink**: The value of the field referenced by the input link retrieved.
- **dbPutLink**: The value of the field referenced by the output link is changed.
- **dbScanPassive**: The record referred to by the forward link is processed if it is passive.

A forward link only makes sense when it refers to a passive record that should be processed when the record containing the link is processed. For input and output links, however, two other attributes can be specified by the application developer, process passive and maximize severity.

5.3.1 Process Passive

Process passive (PP or NPP), is either `TRUE` or `FALSE`. It determines if the linked record should be processed before getting a value from an input link or after writing a value to an output link. The linked record will be processed, via a call to `dbProcess`, only if the record is a passive record and process passive is `TRUE`.

NOTE: Three other options may also be specified: `CA`, `CP`, and `CPP`. These options force the link to be handled like a Channel Access Link. See last section of this chapter for details.

5.3.2 Maximize Severity

Maximize severity (MS or NMS), is `TRUE` or `FALSE`. It determines if alarm severity is propagated across links. For input links the alarm severity of the record referred to by the link is propagated to the record containing the link. For output links the alarm severity of the record containing the link is propagated to the record referred to by the link. In either case, if the severity is changed, the alarm status is set to `LINK_ALARM`.

The method of determining if the alarm status and severity should be changed is called "maximize severity". In addition to its actual status and severity, each record also has a new status and severity. The new status and severity are initially 0, which means `NO_ALARM`. Every time a software component wants to modify the status and severity, it first checks the new severity and only makes a change if the severity it wants to set is greater than the current new severity. If it does make a change, it changes the new status and new severity, not the current status and severity. When database monitors are checked, which is normally done by a record processing routine, the current status and severity are set equal to the new values and the new values reset to zero. The end result is that the current alarm status and severity reflect the highest severity outstanding alarm. If multiple alarms of the same severity are present the status reflects the first one detected.

5.4 Database Locking

The purpose of database locking is to prevent a record from being processed simultaneously by two different tasks. In addition, it prevents "outside" tasks from changing any field while the record is being processed.

The following routines are provided for database locking.

```
dbScanLock(precord);
```

```
dbScanUnlock(precord);
```

The basic idea is to call `dbScanLock` before accessing database records and calling `dbScanUnlock` afterwards. Because of database links (Input, Output, and Forward) a modification to one record can cause modification to other records. Records linked together are placed in the same lock set. `dbScanLock` locks the entire lock set not just the record requested. `dbScanUnlock` unlocks the entire set.

The following rules determine when the lock routines must be called:

1. The periodic, I/O event, and event tasks lock before and unlock after processing:
2. `dbPutField` locks before modifying a record and unlocks afterwards.
3. `dbGetField` locks before reading and unlocks afterwards.
4. Any asynchronous record support completion routine must lock before modifying a record and unlock afterwards.

All records linked via `OUTLINKs` and `FWDLINKs` are placed in the same lock set. Records linked via `INLINKs` with `process_passive` or `maximize_severity TRUE` are also forced to be in the same lock set.

5.5 Database Scanning

Database scanning refers to requests that database records be processed. Four types of scanning are possible:

1. **Periodic** - Records are scanned at regular intervals.
2. **I/O event** - A record is scanned as the result of an I/O interrupt.
3. **Event** - A record is scanned as the result of any task issuing a `post_event` request.
4. **Passive** - A record is scanned as a result of a call to `dbScanPassive`. `dbScanPassive` will issue a record processing request if and only if the record is passive and is not already being processed.

A `dbScanPassive` request results from a task calling one of the following routines:

- **dbScanPassive:** Only record processing routines, `dbGetLink`, `dbPutLink`, and `dbPutField` call `dbScanPassive`. Record processing routines call it for each forward link in the record.
- **dbPutField:** This routine changes the specified field and then, if the field has been declared `process_passive`, calls `dbScanPassive`. Each field of each record type has the attribute `process_passive` declared `TRUE` or `FALSE` in the definition file. This attribute is a global property, i.e. the application developer has no control of it. This use of `process_passive` is used only by `dbPutField`. If `dbPutField` finds the record already active (this can happen to asynchronous records) and it is supposed to cause it to process, it arranges for it to be processed again, when the current processing completes.
- **dbGetLink:** If the link specifies `process passive`, this routine calls `dbScanPassive`. Whether or not `dbScanPassive` is called, it then obtains the specified value.
- **dbPutLink:** This routine changes the specified field. Then, if the link specifies `process passive`, it calls `dbScanPassive`. `dbPutLink` is only called from record processing routines. Note that this usage of `process_passive` is under the control of the application developer. If `dbPutLink` finds the record already active because of a `dbPutField` directed to this record then it arranges for the record to be processed again, when the current processing completes.

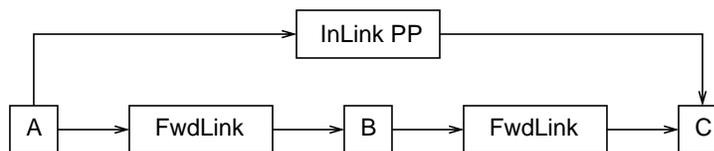
All non-record processing tasks (Channel Access, Sequence Programs, etc.) call `dbGetField` to obtain database values. `dbGetField` just reads values without asking that a record be processed.

5.6 Record Processing

A record is processed as a result of a call to `dbProcess`. Each record support module must supply a routine `process`. This routine does most of the work related to record processing. Since the details of record processing are record type specific this topic is discussed in greater detail in Chapter "Record Support" for details.

5.7 Guidelines for Creating Database Links

The ability to link records together is an extremely powerful feature of the IOC software. In order to use links properly it is important that the Application Developer understand how they are processed. As an introduction consider the following example :



Assume that A, B, and C are all passive records. The notation states that A has a forward link to B and B to C. C has an input link obtaining a value from A. Assume, for some reason, A gets processed. The following sequence of events occurs:

1. A begins processing. While processing a request is made to process B.
2. B starts processing. While processing a request is made to process C.
3. C starts processing. One of the first steps is to get a value from A via the input link.
4. At this point a question occurs. Note that the input link specifies process passive (signified by the `PP` after `InLink`). But process passive states that A should be processed before the value is retrieved. Are we in an infinite loop? The answer is no. Every record contains a field `pact` (processing active), which is set `TRUE` when record processing begins and is not set `FALSE` until all processing completes. When C is processed A still has `pact TRUE` and will not be processed again.
5. C obtains the value from A and completes its processing. Control returns to B.
6. B completes returning control to A
7. A completes processing.

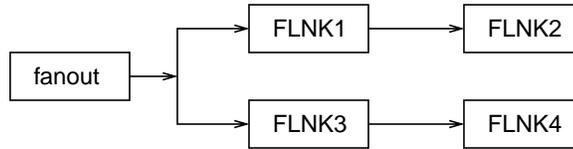
This brief example demonstrates that database links needs more discussion.

5.7.1 Rules Relating to Database Links

5.7.1.1 Processing Order

The processing order is guaranteed to follow the following rules:

1. Forward links are processed in order from left to right and top to bottom. For example the following records are processed in the order `FLNK1`, `FLNK2`, `FLNK3`, `FLNK4` .
 2. If a record has multiple input links (calculation and select records) the input is obtained in the natural order. For example if the fields are named `INPA`, `INPB`, ..., `INPL`, then the links are read in the order A then B then C, etc. Thus if obtaining an input results in a record being processed, the processing order is guaranteed.
-



3. All input and output links are processed before the forward link.

5.7.1.2 Lock Sets

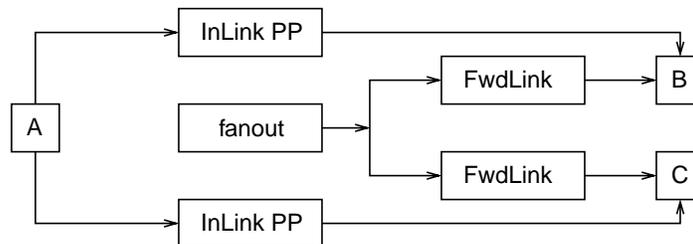
All records, except for the conditions listed in the next paragraph, linked together directly or indirectly are placed in the same lock set. When `dbScanLock` is called the entire set, not just the specified record, is locked. This prevents two different tasks from simultaneously modifying records in the same lock set.

5.7.1.3 PACT - processing active

Each record contains a field `pact`. This field is set `TRUE` at the beginning of record processing and is not set `FALSE` until the record is completely processed. In particular no links are processed with `pact` `FALSE`. This prevents infinite processing loops. The example given at the beginning of this section gives an example. It will be seen in the next two sections that `pact` has other uses.

5.7.1.4 Process Passive: Link option

Input and output links have an option called process passive. For each such link the application developer can specify process passive `TRUE` (`PP`) or process passive `FALSE` (`NPP`). Consider the following example

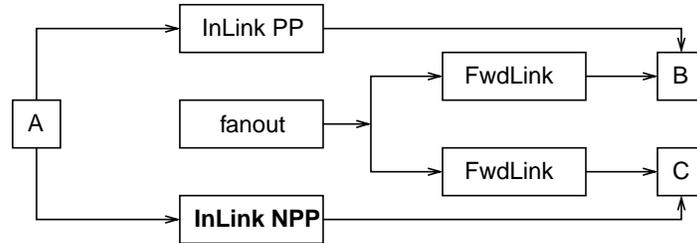


Assume that all records except fanout are passive. When the fanout record is processed the following sequence of events occur:

1. Fanout starts processing and asks that B be processed.
2. B begins processing. It calls `dbGetLink` to obtain data from A.
3. Because the input link has process passive true, a request is made to process A.
4. A is processed, the data value fetched, and control is returned to B
5. B completes processing and control is returned to fanout. Fanout asks that C be processed.
6. C begins processing. It calls `dbGetLink` to obtain data from A.
7. Because the input link has process passive `TRUE`, a request is made to process A.
8. A is processed, the data value fetched, and control is returned to C.
9. C completes processing and returns to fanout

10. The fanout completes

Note that A got processed twice. This is unnecessary. If the input link to C is declared no process passive then A will only be processed once. Thus we should have .



5.7.1.5 Process Passive: Field attribute

Each field of each database record type has an attribute called `process_passive`. This attribute is specified in the record definition file. It is not under the control of the application developer. This attribute is used only by `dbPutField`. It determines if a passive record will be processed after `dbPutField` changes a field in the record. Consult the record specific information in the record reference manual for the setting of individual fields.

5.7.1.6 Maximize Severity: Link option

Input and output links have an option called maximize severity. For each such link the application developer can specify maximize severity TRUE (MS) or maximize severity FALSE (NMS).

When database input or output links are defined, the application developer can specify if alarm severities should be propagated across links. For input links the severity is propagated from the record referred to by the link to the record containing the link. For output links the severity of the record containing the link is propagated to the record referenced by the link. The alarm severity is transferred only if the new severity will be greater than the current severity. If the severity is propagated the alarm status is set equal to `LINK_ALARM`.

5.8 Guidelines for Synchronous Records

A synchronous record is a record that can be completely processed without waiting. Thus the application developer never needs to consider the possibility of delays when he defines a set of related records. The only consideration is deciding when records should be processed and in what order a set of records should be processed.

The following reviews the methods available to the application programmer for deciding when to process a record and for enforcing the order of record processing.

1. A record can be scanned periodically (at one of several rates), via I/O event, or via Event.
2. For each periodic group and for each Event group the phase field can be used to specify processing order.
3. The application programmer has no control over the record processing order of records in different groups.
4. The disable fields (`SDIS`, `DISA`, and `DISV`) can be used to disable records from being processed. By letting the `SDIS` field of an entire set of records refer to the same input record, the entire set can be enabled or disabled simultaneously. See the Record Reference Manual for details.

5. A record (periodic or other) can be the root of a set of passive records that will all be processed whenever the root record is processed. The set is formed by input, output, and forward links.
6. The `process_passive` option specified for each field of each record determines if a passive record is processed when a `dbPutField` is directed to the field. The application developer must be aware of the possibility of record processing being triggered by external sources if `dbPutFields` are directed to fields that have `process_passive TRUE`.
7. The `process_passive` option for input and output links provides the application developer control over how a set of records are scanned.
8. General link structures can be defined. The application programmer should be wary, however, of defining arbitrary structures without carefully analyzing the processing order.

5.9 Guidelines for Asynchronous Records

The previous discussion does not allow for asynchronous records. An example is a GPIB input record. When the record is processed the GPIB request is started and the processing routine returns. Processing, however, is not really complete until the GPIB request completes. This is handled via an asynchronous completion routine. Lets state a few attributes of asynchronous record processing.

During the initial processing for all asynchronous records the following is done:

1. `pact` is set `TRUE`
2. Data is obtained for all input links
3. Record processing is started
4. The record processing routine returns

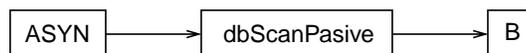
The asynchronous completion routine performs the following algorithm:

5. Record processing continues
6. Record specific alarm conditions are checked
7. Monitors are raised
8. Forward links are processed
9. `pact` is set `FALSE`.

A few attributes of the above rules are:

10. Asynchronous record processing does not delay the scanners.
11. Between the time record processing begins and the asynchronous completion routine completes, no attempt will be made to again process the record. This is because `pact` is `TRUE`. The routine `dbProcess` checks `pact` and does not call the record processing routine if it is `TRUE`. Note, however, that if `dbProcess` finds the record active 10 times in succession, it raises a `SCAN_ALARM`.
12. Forward and output links are triggered only when the asynchronous completion routine completes record processing.

With these rules the following works just fine:

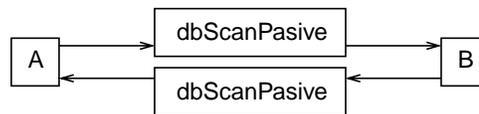


When `dbProcess` is called for record ASYN, processing will be started but `dbScanPassive` will not be called. Until the asynchronous completion routine executes any additional attempts to process ASYN are ignored. When the asynchronous callback is invoked the `dbScanPassive` is performed.

Problems still remain. A few examples are:

5.9.1 Infinite Loop

Infinite processing loops are possible.



Assume both A and B are asynchronous passive records and a request is made to process A. The following sequence of events occur.

1. A starts record processing and returns leaving `pact` TRUE.
2. Sometime later the record completion for A occurs. During record completion a request is made to process B. B starts processing and control returns to A which completes leaving its `pact` field FALSE.
3. Sometime later the record completion for B occurs. During record completion a request is made to process A. A starts processing and control returns to B which completes leaving its `pact` field FALSE.

Thus an infinite loop of record processing has been set up. It is up to the application developer to prevent such loops.

5.9.2 Obtain Old Data

A `dbGetLink` to a passive asynchronous record can get old data.



If A is a passive asynchronous record then the `dbGetLink` request forces `dbProcess` to be called for A. `dbProcess` starts the processing and returns. `dbGetLink` then reads the desired value which is still old because processing will only be completed at a later time.

5.9.3 Delays

Consider the following:



The second ASYN record will not begin processing until the first completes, etc. This is not really a problem except that the application developer must be aware of delays caused by asynchronous records. Again, note that scanners are not delayed, only records downstream of asynchronous records.

5.9.4 Task Abort

If the processing task aborts and the watch dog task cleans up before the asynchronous processing routine completes what happens? If the asynchronous routine completes before the watch dog task runs everything is okay. If it doesn't? This is a more general question of the consequences of having the watchdog timer restart a scan task. EPICS currently does not allow scanners to be automatically restarted.

5.10 *Cached Puts*

The rules followed by `dbPutLink` and `dbPutField` provide for "cached" puts. This is necessary because of asynchronous records. Two cases arise.

The first results from a `dbPutField`, which is a put coming from outside the database, i.e. Channel Access puts. If this is directed to a record that already has `pacT TRUE` because the record started processing but asynchronous completion has not yet occurred, then a value is written to the record but nothing will be done with the value until the record is again processed. In order to make this happen `dbPutField` arranges to have the record reprocessed when the record finally completes processing.

The second case results from `dbPutLink` finding a record already active because of a `dbPutField` directed to the record. In this case `dbPutLink` arranges to have the record reprocessed when the record finally completes processing. If the record is already active because it appears twice in a chain of record processing, it is not reprocessed because the chain of record processing would constitute an infinite loop.

Note that the term caching not queuing is used. If multiple requests are directed to a record while it is active, each new value is placed in the record but it will still only be processed once, i.e. last value wins.

5.11 *putNotify*

`dbPutNotify`, which is called when a Channel Access client calls `ca_put_callback`, is a request to notify the caller when all records processed as a result of the put complete. Because of asynchronous records this can be complicated and the set of records that are processed because of a put may not be deterministic. The result of a `dbPutNotify` is the same as a `dbPutField` except for the following:

- `dbPutNotify`s are queued rather than cached. Thus when additional `dbPutNotify`s are directed to a record that already has an active `dbPutNotify`, they are queued. As each one finishes it releases the next one in the queue.
- If a `dbPutNotify` links to a record that is not active but has a `dbPutNotify` attached to it, then no attempt is made to process the record.

5.12 *Channel Access Links*

A channel access link is:

1. A record link that references a record in a different IOC.
2. A link that the application developer forces to be a channel access link.

A channel access client task (`dbCa`) handles all I/O for channel access links. It does the following:

- At IOC initialization `dbCa` issues channel access search requests for each channel access link.

- For each input link it establishes a channel access monitor. It uses `ca_field_type` and `ca_element_count` when it establishes the monitor. It also monitors the alarm status. Whenever the monitor is invoked the new data is stored in a buffer belonging to `dbCa`. When `iocCore` or the record support module asks for data the data is taken from the buffer and converted to the requested type.
- For each output link, a buffer is allocated the first time `iocCore/record` support issues a put and a channel access connection has been made. This buffer is allocated according to `ca_field_type` and `ca_element_count`. Each time `iocCore/record` support issues a put, the data is converted and placed in the buffer and a request is made to `dbCa` to issue a new `ca_put`.

Even if a link references a record in the same IOC it can be useful to force it to act like a channel access link. In particular the records will not be forced to be in the same lock set. As an example consider a scan record that links to a set of unrelated records, each of which can cause a lot of records to be processed. It is often NOT desirable to force all these records into the same lock set. Forcing the links to be handled as channel access links solves the problem.

CA links which connect between IOCs incur the extra overhead associated with message passing protocols, operating system calls, and network activity. In contrast, CA links which connect records in the same IOC are executed more efficiently by directly calling database access functions such as `dbPutField()` and `dbGetField()`, or by receiving callbacks directly from a database monitor subscription event queue.

Because channel access links interact with the database only via `dbPutField`, `dbGetField`, and a database monitor subscription event queue, their interaction with the database is fundamentally different from database links which are tightly integrated within the code that executes database records. For this reason and because channel access does not understand process passive or maximize severity, the semantics of channel access links are not the same as database links. Let's discuss the channel access semantics of `INLINK`, `OUTLINK`, and `FWDLINK` separately.

5.12.1 INLINK

The options for process passive are:

- **PP** or **NPP** - This link is made a channel access link because the referenced record is not found in the local IOC. It is not possible to honor **PP**, thus the link always acts like **NPP**.
- **CA** - Force the link to be a channel access link.
- **CP** - Force the link to be a channel access link and also request that the record containing the link be processed whenever a monitor occurs.
- **CPP** - Force the link to be a channel access link and also request that the record containing the link, if it is passive, be processed whenever a monitor occurs.

Maximize Severity is honored.

5.12.2 OUTLINK

The options for process passive are:

- **PP** or **NPP** - This link is made a channel access link because the referenced record is not found in the local IOC. It is not possible to honor **PP** thus the link always acts like **NPP**.
- **CA** - Force the link to be a channel access link.

Maximize Severity is not honored.

5.12.3 FWDLINK

A channel access forward link is honored only if it references the `PROC` field of a record. In that case a `ca_put` with a value of 1 is written each time a forward link request is issued.

The options for process passive are:

- **PP** or **NPP** - This link is made a channel access link because the referenced record is not found in the local IOC. It is not possible to honor PP thus it always acts like NPP.
- **CA** - Force the link to be a channel access link.

Maximize Severity is not honored.

Chapter 6: Database Definition

6.1 Overview

This chapter describes database definitions. The following definitions are described:

- Menu
- Record Type
- Device
- Driver
- Registrar
- Variable
- Function
- Breakpoint Table
- Record Instance

Record Instances are fundamentally different from the other definitions. A file containing record instances should never contain any of the other definitions and vice-versa. Thus the following convention is followed:

- **Database Definition File** - A file that contains any type of definition except record instances.
- **Record Instance File** - A file that contains only record instance definitions.

This chapter also describes utility programs which operate on these definitions

Any combination of definitions can appear in a single file or in a set of files related to each other via include files.

6.2 Brief Summary of Database Definition Syntax

```
path "path"
addpath "path"
include "filename"
#comment
menu(name) {
    include "filename"
    choice(choice_name, "choice_value")
    ...
}

recordtype(record_type) {
    include "filename"
    field(field_name, field_type) {
        asl(asl_level)
        initial("init_value")
    }
}
```

```
    promptgroup(gui_group)
    prompt("prompt_value")
    special(special_value)
    pp(pp_value)
    interest(interest_level)
    base(base_type)
    size(size_value)
    extra("extra_info")
    menu(name)
}
...
}

device(record_type,link_type,dset_name,"choice_string")

driver(drvet_name)

registrar(function_name)

variable(variable_name)

breaktable(name) {
    raw_value eng_value
    ...
}
```

#The Following defines a Record Instance

```
record(record_type,record_name) {
    include "filename"
    field(field_name,"value")
    ...
}
```

#NOTE: GDCT uses the keyword grecord instead of record

6.3 General Rules for Database Definition

6.3.1 Keywords

The following are keywords, i.e. they may not be used as values unless they are enclosed in quotes:

```
path
addpath
include
menu
choice
recordtype
field
device
driver
```

```

  registrar
  variable
  breaktable
  record
  grecord
  
```

6.3.2 Unquoted Strings

In the summary section, some values are shown as quoted strings and some unquoted. The actual rule is that any string consisting of only the following characters does not have to be quoted:

```
a-z A-Z 0-9 _ - : . [ ] < > ;
```

These are also the legal characters for process variable names. Thus in many cases quotes are not needed.

6.3.3 Quoted Strings

A quoted string can contain any ascii character except the quote character ". The quote character itself can given by using \ as an escape. For example "\" is a quoted string containing the single character ".

6.3.4 Macro Substitution

Macro substitutions are permitted inside quoted strings. The macro has the form:

```

  $(name)
  or
  ${name}
  
```

6.3.5 Escape Sequences

Except for \ " the database routines never translate standard C escape sequences, however, dbTranslateEscape can be used to translate the standard C escape sequences:

```
\a \b \f \n \r \t \v \\ \? \' \" \000 \xhh
```

(\000 represent an octal number of 1, 2, or 3 digits. \xhh represents a hexadecimal number of 1 or 2 digits) A typical use is device support which expects escape sequences in the parm field:

The routine is:

```

  int dbTranslateEscape(char *s,const char *ct);
  /*
   * copies ct to s while substituting escape sequences
   * returns the length of the resultant string
   * The result may contain 0 characters
   */
  
```

6.3.6 Define before referencing

No item can be referenced until it is defined. For example a recordtype menu field can not reference a menu unless that menu definition has already been defined. Another example is that a record instance can not appear until the associated record type has been defined.

6.3.7 Multiple Definitions

If a particular menu, recordtype, device, driver, or breakpoint table is defined more than once, then only the first instance is used. Record instance definitions are cumulative, i.e. each time a new field value is encountered it replaces the previous value.

6.3.8 filename extension

By convention:

- Record instances files have the extension ".db"
- Database definition files have the extension ".dbd".

6.3.9 path addpath

The path follows the standard Unix convention, i.e. it is a list of directory names separated by colons (Unix) or semicolons (winXX).

Format:

```
path "dir:dir...:dir"
addpath "dir:dir...:dir"
```

NOTE: In winXX the separator is ; instead of :

The path command specifies the current path. The addpath appends directory names to the current path. The path is used to locate the initial database file and included files. An empty dir at the beginning, middle, or end of a non-empty path string means the current directory. For example:

```
nnn::mmm      # Current directory is between nnn and mmm
:nnn          # Current directory is first
nnn:          # Current directory is last
```

Utilities which load database files (dbExpand, dbLoadDatabase, etc.) allow the user to specify an initial path. The path and addpath commands can be used to change or extend the initial path.

The initial path is determined as follows:

- If an initial path is specified, it is used. Else:
- If the environment variable EPICS_DB_INCLUDE_PATH is defined, it is used. Else:
- the default path is ".", i.e. the current directory.

The path is used unless the filename contains a / or \. The first directory containing the specified file is used.

6.3.10 include

Format:

```
include "filename"
```

An include statement can appear at any place shown in the summary. It uses the path as specified above.

6.3.11 comment

The comment symbol is "#". Whenever the comment symbol appears, it and all characters through the end of the line are ignored.

6.4 Menu

Format:

```
menu(name) {
    choice(choice_name, "choice_value")
    ...
}
```

Where:

name - Name for menu. This is the unique name identifying the menu. If duplicate definitions are specified, only the first is used.

choice_name - The name placed in the enum generated by dbToMenuH or dbToRecordtypeH

choice_value - The value associated with the choice.

Example:

```
menu(menuYesNo) {
    choice(menuYesNoNO, "NO")
    choice(menuYesNoYES, "YES")
}
```

6.5 Record Type

6.5.1 Format:

```
recordtype(record_type) {
    field(field_name, field_type) {
        asl(asl_level)
        initial("init_value")
        promptgroup(gui_group)
        prompt("prompt_value")
        special(special_value)
        pp(pp_value)
        interest(interest_level)
        base(base_type)
        size(size_value)
        extra("extra_info")
        menu("name")
    }
    ...
}
```

6.5.2 rules

- **asl** - Access Security Level. The default is ASL1. Access Security is discussed in a later chapter. Only two values are permitted for this field (ASL0 and ASL1). Fields which operators normally change are assigned ASL0. Other

fields are assigned ASL1. For example, the VAL field of an analog output record is assigned ASL0 and all other fields ASL1. This is because only the VAL field should be modified during normal operations.

- **initial** - Initial Value.
- **promptgroup** - Prompt group to which field belongs. This is for use by Database Configuration Tools. This is defined only for fields that can be given values by database configuration tools. File `guigroup.h` contains all possible definitions. The different groups allow database configuration tools to present the user with groups of fields rather than all prompt fields. I don't know of any tool that currently uses groups.
- **prompt** - A prompt string for database configuration tools. Optional if `promptgroup` is not defined.
- **special** - If specified, then special processing is required for this field at run time.
- **pp** - Should a passive record be processed when Channel Access writes to this field? The default is NO.
- **interest** - Only used by the `dbpr` shell command.
- **base** - For integer fields, a base of DECIMAL or HEX can be specified. The default is DECIMAL.
- **size** - Must be specified for `DBF_STRING` fields.
- **extra** - Must be specified for `DBF_NOACCESS` fields.
- **menu** - Must be specified for `DBF_MENU` fields. It is the name of the associated menu.

6.5.3 definitions

- **record_type** - The unique name of the record type. If duplicates are specified, only the first definition is used.
 - **field_name** - The field name. Only alphanumeric characters are allowed. When include files are generated, the field name is converted to lower case. Previous versions of EPICS required that field name be a maximum of four characters. Although this restriction no longer exists, problems may arise with some Channel Access clients if longer field names are chosen.
 - **field_type** - This must be one of the following values:
 - `DBF_STRING`
 - `DBF_CHAR`
 - `DBF_UCHAR`
 - `DBF_SHORT`
 - `DBF_USHORT`
 - `DBF_LONG`
 - `DBF_ULONG`
 - `DBF_FLOAT`
 - `DBF_DOUBLE`
 - `DBF_ENUM`
 - `DBF_MENU`
 - `DBF_DEVICE`
 - `DBF_INLINK`
 - `DBF_OUTLINK`
 - `DBF_FWDLINK`
 - `DBF_NOACCESS`
 - **asl_level** - This must be one of the following values:
 - ASL0
 - ASL1 (default value)
 - **init_value** - A legal value for data type.
 - **prompt_value** - A prompt value for database configuration tools.
 - **gui_group** - This must be one of the following:
-

- GUI_COMMON
- GUI_ALARMS
- GUI_BITS1
- GUI_BITS2
- GUI_CALC
- GUI_CLOCK
- GUI_COMPRESS
- GUI_CONVERT
- GUI_DISPLAY
- GUI_HIST
- GUI_INPUTS
- GUI_LINKS
- GUI_MBB
- GUI_MOTOR
- GUI_OUTPUT
- GUI_PID
- GUI_PULSE
- GUI_SELECT
- GUI_SEQ1
- GUI_SEQ2
- GUI_SEQ3
- GUI_SUB
- GUI_TIMER
- GUI_WAVE
- GUI_SCAN

NOTE: GUI types were invented with the intention of allowing database configuration tools to prompt for groups of fields and when a user selects a group the fields within the group. Since this feature has seldom been used, many record types have not assigned the correct GUI groups to some fields.

- **special_value** must be one of the following:

- An integer value greater than 103. In this case, the record support special routine is called whenever the field is modified by database access. This feature is present only for compatibility. New support modules should use SPC_MOD.

The following value disallows access to field.

- SPC_NOMOD - This means that field can not be modified at runtime except by the record/device support modules for the record type.

The following values are used for database common. They must NOT be used for record specific fields.

- SPC_SCAN - Scan related field.
- SPC_ALARMACK - Alarm acknowledgment field.
- SPC_AS - Access security field.

The following value is used if record support wants to trap dbNameToAddr calls.

- SPC_DBADDR - This is set if the record support cvt_dbaddr routine should be called whenever dbNameToAddr is called, i.e. when code outside record/device support want to access the field.

The following values all result in the record support special routine being called whenever database access modifies the field. The only reason for multiple values is that originally it seemed like a good idea. New support modules should only use `SPC_MOD`.

- `SPC_MOD` - Notify when modified, i.e. call the record support special routine whenever the field is modified by database access.
- `SPC_RESET` - a reset field is being modified.
- `SPC_LINCONV` - A linear conversion field is being modified.
- `SPC_CALC` - A calc field is being modified.
- **pp_value** - Should a passive record be processed when Channel Access writes to this field? The allowed values are:
 - NO (default)
 - YES
- **interest_level** - An interest level for the `dbpr` command.
- **base** - For integer type fields, the default base. The legal values are:
 - DECIMAL (Default)
 - HEX
- **size_value** - The number of characters for a `DBF_STRING` field.
- **extra_info** - For `DBF_NOACCESS` fields, this is the C language definition for the field. The definition must end with the fieldname in lower case.

6.5.4 Example

The following is the definition of the binary input record.

```
recordtype(bi) {
  include "dbCommon.dbd"
  field(INP,DBF_INLINK) {
    prompt("Input Specification")
    promptgroup(GUI_INPUTS)
    interest(1)
  }
  field(VAL,DBF_ENUM) {
    prompt("Current Value")
    promptgroup(GUI_INPUTS)
    asl(ASL0)
    pp(TRUE)
  }
  field(ZSV,DBF_MENU) {
    prompt("Zero Error Severity")
    promptgroup(GUI_ALARMS)
    pp(TRUE)
    interest(1)
    menu(menuAlarmSevr)
  }
  field(OSV,DBF_MENU) {
    prompt("One Error Severity")
    promptgroup(GUI_BITS1)
    pp(TRUE)
    interest(1)
    menu(menuAlarmSevr)
  }
}
```

```

}
field(COSV,DBF_MENU) {
    prompt("Change of State Svr")
    promptgroup(GUI_BITS2)
    pp(TRUE)
    interest(1)
    menu(menuAlarmSevr)
}
field(ZNAM,DBF_STRING) {
    prompt("Zero Name")
    promptgroup(GUI_CALC)
    pp(TRUE)
    interest(1)
    size(20)
}
field(ONAM,DBF_STRING) {
    prompt("One Name")
    promptgroup(GUI_CLOCK)
    pp(TRUE)
    interest(1)
    size(20)
}
field(RVAL,DBF_ULONG) {
    prompt("Raw Value")
    pp(TRUE)
}
field(ORAW,DBF_ULONG) {
    prompt("prev Raw Value")
    special(SPC_NOMOD)
    interest(3)
}
field(MASK,DBF_ULONG) {
    prompt("Hardware Mask")
    special(SPC_NOMOD)
    interest(1)
}
field(LALM,DBF_USHORT) {
    prompt("Last Value Alarmed")
    special(SPC_NOMOD)
    interest(3)
}
field(MLST,DBF_USHORT) {
    prompt("Last Value Monitored")
    special(SPC_NOMOD)
    interest(3)
}
field(SIOL,DBF_INLINK) {
    prompt("Sim Input Specifctn")
    promptgroup(GUI_INPUTS)
    interest(1)
}
field(SVAL,DBF_USHORT) {
    prompt("Simulation Value")
}

```

```

    }
    field(SIML,DBF_INLINK) {
        prompt("Sim Mode Location")
        promptgroup(GUI_INPUTS)
        interest(1)
    }
    field(SIMM,DBF_MENU) {
        prompt("Simulation Mode")
        interest(1)
        menu(menuYesNo)
    }
    field(SIMS,DBF_MENU) {
        prompt("Sim mode Alarm Svrty")
        promptgroup(GUI_INPUTS)
        interest(2)
        menu(menuAlarmSevr)
    }
}

```

6.6 Device

6.6.1 Format:

```

device(record_type,link_type,dset_name,"choice_string")
...

```

6.6.2 definitions

- **record_type** - Record type. The combination of `record_type` and `choice_string` must be unique. If the same combination appears multiple times, the first definition is used.
- **link_type** - Link type. This must be one of the following:
 - CONSTANT
 - PV_LINK
 - VME_IO
 - CAMAC_IO
 - AB_IO
 - GPIB_IO
 - BITBUS_IO
 - INST_IO
 - BBGPIB_IO
 - RF_IO
 - VXI_IO
- **dset_name** - The exact name of the device support entry table without the trailing "DSET". Duplicates are not allowed.
- **choice_string** - Choice string for database configuration tools. Note that it must be enclosed in "". Note that for a given record type, each `choice_string` must be unique.

6.6.3 Examples

```
device(ai,CONSTANT,devAiSoft,"Soft Channel")
device(ai,VME_IO,devAiXy566Se,"XYCOM-566 SE Scanned")
```

6.7 Driver

6.7.1 Format:

```
driver(drvet_name)
```

6.7.2 Definitions

- **drvet_name** - If duplicates are defined, only the first is used.

6.7.3 Examples

```
driver(drvVxi)
driver(drvXy210)
```

6.8 Registrar Declaration

6.8.1 Format:

```
registrar(function_name)
```

6.8.2 Definitions

- **function_name** - The name of an C function that accepts no arguments, returns void and has been exported from its source file with an `epicsExportRegistrar` declaration, e.g.

```
static void myRegistrar(void);
epicsExportRegistrar(myRegistrar);
```

This can be used to register functions for use by subroutine records or that can be invoked from `iocsh`. The example application described in Chapter 2.2, "Example IOC Application" on page 11" provides an example of how to register functions for subroutine records.

6.8.3 Example

```
registrar(asSub)
```

6.9 Variable Declaration

6.9.1 Format:

```
variable(variable_name[, type])
```

6.9.2 Definitions

- **variable_name** - The name of a C variable which has been exported from its source file with an `epicsExportAddress` declaration.
- **type** - The C variable type. If not present, **int** is assumed. Currently only **int** and **double** variables are supported.

This registers a diagnostic/configuration variable for device and driver support or a subroutine record subroutine so that the variable can be read and set with the `iocsh var` command (Section on page 247). The example application described in Section 2.2 on page 11 provides an example of how to register a debug variable for a subroutine record.

6.9.3 Example

In an application C source file:

```
#include <epicsExport.h>
.
.
static double myParameter;
epicsExportAddress(double, myParameter);
```

In an application database definition file:

```
variable(myParameter, double)
```

6.10 Function Declaration

6.10.1 Format:

```
function(function_name)
```

6.10.2 Definitions

- **function_name** - The name of a C function which has been exported from its source file with an `epicsRegisterFunction` declaration.

This registers a function so that it can be found in the function registry for use by records which refer to the function by name. The example application described in Chapter 2.2, “Example IOC Application” on page 11 provides an example of how to register functions for a subroutine record.

6.10.3 Example

In an application C source file:

```
#include <epicsExport.h>
.
.
static long myFunction(void *argp) {
.
.
}
epicsRegisterFunction(myFunction);
```

In an application database definition file:

```
function(myFunction
```

6.11 Breakpoint Table

6.11.1 Format:

```
breaktable(name) {
    raw_value eng_value
    ...
}
```

6.11.2 Definitions

- **name** - Name, which must be alpha-numeric, of the breakpoint table. If duplicates are specified the first is used.
- **raw_value** - The raw value, i.e. the actual ADC value associated with the beginning of the interval.
- **eng_value** - The engineering value associated with the beginning of the interval.

6.11.3 Example

```
breaktable(typeJdegC) {
    0.000000 0.000000
    365.023224 67.000000
    1000.046448 178.000000
    3007.255859 524.000000
    3543.383789 613.000000
    4042.988281 692.000000
    4101.488281 701.000000
}
```

6.12 Record Instance

6.12.1 Format:

```
record(record_type, record_name) {  
    field(field_name, "field_value")  
    info(info_name, "info_value")  
    ...  
}
```

6.12.2 definitions

- **record_type** - The record type.
- **record_name** - The record name. This must be composed of the following characters:
a-z A-Z 0-9 _ - + : [] < > ;
NOTE: If macro substitutions are used the name must be quoted.
If duplicate definitions are given for the same record, then the last value given for each field is the value assigned to the field.
- **field_name** - A field name
- **field_value** - Depends on field type. Inside the double quotes the field value string may contain escaped C89 characters such as \", \t, \n, \064 and \x7e, and these will be translated appropriately when loading the database.
 - DBF_STRING
Any ASCII string. If it exceeds the field length, it will be truncated.
 - DBF_CHAR, DBF_UCHAR, DBF_SHORT, DBF_USHORT, DBF_LONG, DBF_ULONG
A string that represents a valid integer. The standard C conventions are applied, i.e. a leading 0 means the value is given in octal and a leading 0x means that value is given in hex.
 - DBF_FLOAT, DBF_DOUBLE
The string must represent a valid floating point number.
 - DBF_MENU
The string must be one of the valid choices for the associated menu.
 - DBF_DEVICE
The string must be one of the valid device choice strings.
 - DBF_INLINK, DBF_OUTLINK
NOTES:
 - In the field is INP or OUT then it is associated with field DTYP. Other DBF_INLINK and DBF_OUTLINK fields can be either CONSTANT or PV_LINKs
 - DTYP must be defined before the associated INP or OUT field.
 - Choosing the DTYP implicitly chooses a bus type.
 - A DTYP of CONSTANT can be either a constant or a PV_LINK.

The allowed value depends on the bus type as follows:

- CONSTANT
A constant valid for the field associated
- PV_LINK
A value of the form:

record.field process maximize

field, process, and maximize are optional.

The default value for `field` is `VAL`.

`process` can have one of the following values:

- `NPP` - No Process Passive (Default)
- `PP` - Process Passive
- `CA` - Force link to be a channel access link
- `CP` - CA and process on monitor
- `CPP` - CA and process on monitor if record is passive

NOTES:

`CP` and `CPP` are valid only for `INLINKs`.

`FWD_LINKs` can be `PP` or `CA`. If a `FWD_LINK` is a channel access link it must reference the `PROC` field.

`maximize` can have one of the following values

- `NMS` - No Maximize Severity (Default)
- `MS` - Maximize severity

- `VME_IO`

`#Ccard Ssignal @parm`

where:

`card` - the card number of associated hardware module.

`signal` - signal on card

`parm` - An arbitrary character string of up to 31 characters.

This field is optional and is device specific.

- `CAMAC_IO`

`#Bbranch Ccrate Nstation Asubaddress Ffunction @parm`

`branch`, `crate`, `station`, `subaddress`, and `function` should be obvious to `camac` users.

`subaddress` and `function` are optional (0 if not given). `parm` is also optional and is device dependent (25 characters max).

- `AB_IO`

`#Llink Aadapter Ccard Ssignal @parm`

`link` - Scanner, i.e. `vme` scanner number

`adapter` - Adapter. Allen Bradley also calls this rack

`card` - Card within Allen Bradley Chassis

`signal` - signal on card

`parm` - An optional character string that is device dependent(27 char max)

- `GPIB_IO`

`#Llink Aaddr @parm`

`link` - `gpiib` link, i.e. interface

`addr` - `GPIB` address

`parm` - device dependent character string (31 char max)

- `BITBUS_IO`

`#Llink Nnode Pport Ssignal @parm`

`link` - link, i.e. `vme` bitbus interface.

`node` - bitbus node

`port` - port on the node

`signal` - signal on port

`parm` - device specific character string(31 char max)

- `INST_IO`

`@parm`

`parm` - Device dependent character string(35 char max)

- `BGGPIB_IO`

`#Llink Bbbaddr Ggpibaddr @parm`

`link` - link, i.e. `vme` bitbus interface.

`bbaddr` - bitbus address

- gpibaddr - gpib address
- parm - optional device dependent character string(31 char max)
- RF_IO
 - #Rcryo Mmicro Ddataset Eelement
- VXI_IO
 - #Vframe Cslot Ssignal @parm (Dynamic addressing)
 - or
 - #Vla Signal @parm (Static Addressing)
 - frame - VXI frame number
 - slot - Slot within VXI frame
 - la - Logical Address
 - signal - Signal Number
 - parm - device specific character string(25 char max)
- DBF_FWDLINK
 - This is either not defined or else is a PV_LINK. See above for definitions.
- **info_name** - The name of an information item related to this record. See section 6.13 below for more on information items.
- **info_value** - Any ASCII string. IOC applications using this information item may place additional restrictions on the contents of the string.

6.12.3 Examples

```

record(ai, STS_AbAiMaS0) {
  field(SCAN, ".1 second")
  field(DTYP, "AB-1771IFE-4to20MA")
  field(INP, "#L0 A2 C0 S0 F0 @")
  field(PREC, "4")
  field(LINR, "LINEAR")
  field(EGUF, "20")
  field(EGUL, "4")
  field(EGU, "MilliAmps")
  field(HOPR, "20")
  field(LOPR, "4")
}
record(ao, STS_AbAoMaC1S0) {
  field(DTYP, "AB-1771OFE")
  field(OUT, "#L0 A2 C1 S0 F0 @")
  field(LINR, "LINEAR")
  field(EGUF, "20")
  field(EGUL, "4")
  field(EGU, "MilliAmp")
  field(DRVH, "20")
  field(DRVL, "4")
  field(HOPR, "20")
  field(LOPR, "4")
  info(autosaveFields, "VAL")
}
record(bi, STS_AbDiA0C0S0) {
  field(SCAN, "I/O Intr")
  field(DTYP, "AB-Binary Input")
  field(INP, "#L0 A0 C0 S0 F0 @")
  field(ZNAM, "Off")
}

```

```
    field(ONAM, "On" )  
}
```

6.13 Record Information Item

Information items provide a way to attach named string values to individual record instances that are loaded at the same time as the record definition. They can be attached to any record without having to modify the record type, and can be retrieved by programs running on the IOC (they are not visible via Channel Access at all). Each item attached to a single record must have a unique name by which it is addressed, and database access provides routines to allow a record's info items to be scanned, searched for, retrieved and set. At runtime a `void*` pointer can also be associated with each item, although only the string value can be initialized from the record definition when the database is loaded.

6.14 Record Attribute

Each record type can have a set of record attributes. Each attribute is a “psuedo” field that can be accessed via database and channel access. An attribute is given a name that acts like a field name which has the same value for every instance of the record type. Two attributes are generated automatically for each record type: `RTYP` and `VERS`. The value for `RTYP` is the record type name. The default value for `VERS` is "none specified", which can be changed by record support. Record support can call the following routine to create new attributes or change existing attributes:

```
long dbPutAttribute(char *recordTypename,  
                  char *name, char*value)
```

The arguments are:

`recordTypename` - The name of recordtype.
`name` - The attribute name, i.e. the psuedo field name.
`value` - The value assigned to the attribute.

6.15 Breakpoint Tables - Discussion

The menu `menuConvert` is used for field `LINR` of the `ai` and `ao` records. These records allow raw data to be converted to/from engineering units via one of the following:

1. No Conversion.
2. Slope Conversion.
3. Linear Conversion.
4. Breakpoint table.

Other record types can also use this feature. The first choice specifies no conversion; the second and third are both linear conversions, the difference being that for Slope conversion the user specifies the conversion slope and offset directly, whereas for Linear conversions these are calculated by the device support from the requested Engineering Units range and the device support's knowledge of the hardware conversion range. The remaining choices are assumed to be the names of breakpoint tables. If a breakpoint table is chosen, the record support modules call `cvtRawToEngBpt` or `cvtEngToRawBpt`. You can look at the `ai` and `ao` record support modules for details.

If a user wants to add additional breakpoint tables, then the following should be done:

Chapter 6: Database Definition

Breakpoint Tables - Discussion

- Copy the menuConvert.dbd file from EPICS base/src/bpt
- Add definitions for new breakpoint tables to the end
- Make sure modified menuConvert.dbd is loaded into the IOC instead of EPICS version.

It is only necessary to load a breakpoint file if a record instance actually chooses it. It should also be mentioned that the Allen Bradley IXE device support misuses the LINR field. If you use this module, it is very important that you do not change any of the EPICS supplied definitions in menuConvert.dbd. Just add your definitions at the end.

If a breakpoint table is chosen, then the corresponding breakpoint file must be loaded into the IOC before iocInit is called.

Normally, it is desirable to directly create the breakpoint tables. However, sometimes it is desirable to create a breakpoint table from a table of raw values representing equally spaced engineering units. A good example is the Thermocouple tables in the OMEGA Engineering, INC Temperature Measurement Handbook. A tool makeBpt is provided to convert such data to a breakpoint table.

The format for generating a breakpoint table from a data table of raw values corresponding to equally spaced engineering values is:

```
!comment line
<header line>
<data table>
```

The header line contains the following information:

- **Name:** An alphanumeric ascii string specifying the breakpoint table name
- **Low Value Eng:** Engineering Units Value for first breakpoint table entry
- **Low Value Raw:** Raw value for first breakpoint table entry
- **High Value Eng:** Engineering Units: Highest Value desired
- **High Value Raw:** Raw Value for High Value Eng
- **Error:** Allowed error (Engineering Units)
- **First Table:** Engineering units corresponding to first data table entry
- **Last Table:** Engineering units corresponding to last data table entry
- **Delta Table:** Change in engineering units per data table entry

An example definition is:

```
"TypeKdegF" 32 0 1832 4095 1.0 -454 2500 1
<data table>
```

The breakpoint table can be generated by executing

```
makeBpt bptXXX.data
```

The input file must have the extension of data. The output filename is the same as the input filename with the extension of dbd.

Another way to create the breakpoint table is to include the following definition in a Makefile.Vx:

```
BPTS += bptXXX.dbd
```

NOTE: This requires the naming convention that all data tables are of the form bpt<name>.data and a breakpoint table bpt<name>.dbd.

6.16 Menu and Record Type Include File Generation.

6.16.1 Introduction

Given a file containing menus, `dbToMenuH` generates an include file that can be used by any code which uses the associated menus. Given a file containing any combination of menu definitions and record type definitions, `dbToRecordtypeH` generates an include file that can be used by any code which uses the menus and record type.

EPICS base uses the following conventions for managing menu and recordtype definitions. Users generating local record types are encouraged to do likewise.

- Each menu that is either for fields in database common (for example `menuScan`) or is of global use (for example `menuYesNo`) is defined in a separate file. The name of the file is the same as the menu name with an extension of `.dbd`. The name of the generated include file is the menu name with an extension of `.h`. Thus `menuScan` is defined in a file `menuScan.dbd` and the generated include file is named `menuScan.h`
- Each record type definition is defined in a separate file. In addition, this file contains any menu definitions that are used only by that record type. The name of the file is the same as the recordtype name followed by `Record.dbd`. The name of the generated include file is the same name with an extension of `.h`. Thus `aoRecord` is defined in a file `aoRecord.dbd` and the generated include file is named `aoRecord.h`. Since `aoRecord` has a private menu called `aoOIF`, the `dbd` file and the generated include file have definitions for this menu. Thus for each record type, there are two source files (`xxxRecord.dbd` and `xxxRecord.c`) and one generated file (`xxxRecord.h`).

Before continuing, it should be mentioned that Application Developers don't have to execute `dbToMenuH` or `dbToRecordtypeH`. If a developer uses the proper naming conventions, it is only necessary to add definitions to their `Makefile`. Consult the chapter on the EPICS Build Facility for details..

6.16.2 dbToMenuH

This tool is executed as follows:

```
dbToMenuH -Idir -Smacsub menuXXX.dbd
```

It generates a file which has the same name as the input file but with an extension of `.h`. Multiple `-I` options can be specified for an include path and multiple `-S` options for macro substitution.

For example `menuPriority.dbd`, which contains the definitions for processing priority contains:

```
menu(menuPriority) {  
    choice(menuPriorityLOW, "LOW")  
    choice(menuPriorityMEDIUM, "MEDIUM")  
    choice(menuPriorityHIGH, "HIGH")  
}
```

The include file, `menuPriority.h`, generated by `dbToMenuH` contains:

```
#ifndef INCMenuPriorityH  
#define INCMenuPriorityH  
typedef enum {  
    menuPriorityLOW,  
    menuPriorityMEDIUM,  
    menuPriorityHIGH,  
}menuPriority;  
#endif /*INCMenuPriorityH*/
```

Any code that needs to use the priority menu values should use these definitions.

6.16.3 dbToRecordtypeH

This tool is executed as follows:

```
dbToRecordtypeH -Idir -Smacsub xxxRecord.dbd
```

It generates a file which has the same name as the input file but with an extension of h. Multiple `-I` options can be specified for an include path and multiple `-S` options for macro substitution.

For example `aoRecord.dbd`, which contains the definitions for the analog output record contains:

```
menu(aoOIF) {
    choice(aoOIF_Full, "Full")
    choice(aoOIF_Incremental, "Incremental")
}
recordtype(ao) {
    include "dbCommon.dbd"
    field(VAL,DBF_DOUBLE) {
        prompt("Desired Output")
        asl(ASL0)
        pp(TRUE)
    }
    field(OVAL,DBF_DOUBLE) {
        prompt("Output Value")
    }
    ... (Many more field definitions)
}
}
```

The include file, `aoRecord.h`, generated by `dbToRecordtypeH` contains:

```
#include "ellLib.h"
#include "epicsMutex.h"
#include "link.h"
#include "epicsTime.h"
#include "epicsTypes.h"

#ifndef INCaoOIFH
#define INCaoOIFH
typedef enum {
    aoOIF_Full,
    aoOIF_Incremental,
}aoOIF;
#endif /*INCaoOIFH*/
#ifndef INCaoH
#define INCaoH
typedef struct aoRecord {
    char          name[29]; /*Record Name*/
    ... Remaining fields in database common
    double       val;      /*Desired Output*/
    double       oval;     /*Output Value*/
    ... remaining record specific fields
} aoRecord;
```

```

#define aoRecordNAME      0
... defines for remaining fields in database common
#define aoRecordVAL       42
#define aoRecordOVAL     43
... defines for remaining record specific fields
#ifdef GEN_SIZE_OFFSET
int aoRecordSizeOffset(dbRecordType *pdbRecordType)
{
    aoRecord *prec = 0;
    pdbRecordType->papFldDes[0]->size=sizeof(prec->name);
    pdbRecordType->papFldDes[0]->offset=
        (short)((char *)&prec->name - (char *)prec);
    ... code to compute size&offset for other fields in dbCommon
    pdbRecordType->papFldDes[42]->size=sizeof(prec->val);
    pdbRecordType->papFldDes[42]->offset=
        (short)((char *)&prec->val - (char *)prec);
    pdbRecordType->papFldDes[43]->size=sizeof(prec->oval);
    pdbRecordType->papFldDes[43]->offset=
        (short)((char *)&prec->oval - (char *)prec);
    ... code to compute size&offset for remaining fields
    pdbRecordType->rec_size = sizeof(*prec);
    return(0);
}
#endif /*GEN_SIZE_OFFSET*/

```

The analog output record support module and all associated device support modules should use this include file. No other code should use it. Let's discuss the various parts of the file.:

- The enum generated from the menu definition should be used to reference the value of the field associated with the menu.
- The typedef and structure defining the record are used by record support and device support to access fields in an analog output record.
- A #define is present for each field within the record. This is useful for the record support routines that are passed a pointer to a DBADDR structure. They can have code like the following:

```

switch (dbGetFieldIndex(pdbAddr)) {
    case aoRecordVAL :
        ...
        break;
    case aoRecordXXX:
        ...
        break;
    default:
        ...
}

```

The C source routine `aoRecordSizeOffset` is automatically called when a record type file is loaded into an IOC. Thus user code does not have to be aware of this routine except for the following convention: The associate record support module **MUST** include the statements:

```

#define GEN_SIZE_OFFSET
#include "xxxRecord.h"
#undef GEN_SIZE_OFFSET

```

This convention ensures that the routine is defined exactly once.

6.17 dbExpand

```
dbExpand -Idir -Smacsub -ooutfile file1 file2 ...
```

Multiple `-I` options can be specified for an include path, and multiple `-S` options for macro substitution. If no output filename is specified with `-ooutfile` then the output will go to `stdout`. Note that the environment variable `EPICS_DB_INCLUDE_PATH` can also be used in place of the `-I` options.

NOTE: This is supported only on the host.

This command reads all the input files and then writes a file containing the definitions for all information described by the input files. The output content differs from the input in that comment lines do not appear and all include files are expanded.

This routine is extremely useful if an IOC is not using NFS for the `dbLoadDatabase` commands. It takes more than 2 minutes to load the `base/rec/base.dbd` file into an IOC if NFS is not used. If `dbExpand` creates a local `base.dbd` file, it takes about 7 seconds to load (25 MHZ 68040 IOC).

6.18 dbLoadDatabase

```
dbLoadDatabase(char *dbdfile, char *path, char *substitutions)
```

NOTES:

- IOC Only
- Using a path on a vxWorks ioc does not work very well.
- Both path and substitutions can be null.
- `dbdfile` may contain environment variable macros of the form `${MOTOR}` which will be expanded before the file is opened.

This command loads a database file containing any of the definitions given in the summary at the beginning of this chapter.

As each line of `dbdfile` is read, the substitutions specified in `substitutions` is performed. The substitutions are specified as follows:

```
"var1=sub1,var2=sub3,..."
```

Variables are specified in the `dbfile` as `$(variable_name)`. If the substitution string

```
"a=1,b=2,c=\"this is a test\""
```

were used, any variables `$(a)`, `$(b)`, `$(c)` would be substituted with the appropriate data.

6.18.1 EXAMPLE

For example, let `test.db` be:

```
record(ai,"$(pre)testrec1")
record(ai,"$(pre)testrec2")
record(stringout,"$(pre)testrec3") {
```

```
    field(VAL, "${STR}")
    field(SCAN, "${SCAN}")
}
```

Then issuing the command:

```
dbLoadDatabase("test.db", 0, "pre=TEST,STR=test,SCAN=Passive")
```

gives the same results as loading:

```
record(ai, "TESTtestrec1")
record(ai, "TESTtestrec2")
record(stringout, "TESTtestrec3") {
    field(VAL, "test")
    field(SCAN, "Passive")
}
```

6.19 dbLoadRecords

```
dbLoadRecords(char* dbfile, char* substitutions)
```

NOTES:

- IOC Only.
- dbfile must contain only record instances.
- dbfile may contain environment variable macros of the form `${MOTOR}` which will be expanded before the file is opened.

6.20 dbLoadTemplate

```
dbLoadTemplate(char* template_def)
```

NOTES:

- IOC Only.
- MSI can be used to expand templates on the host.

`dbLoadTemplate` reads a template definition file. This file contains rules about loading database instance files, which contain `$(xxx)` macros, and performing substitutions.

`template_def` contains the rules for performing substitutions on the instance files. For convenience two formats are provided. The format is either:

```
file name.template {
    { var1=sub1_for_set1, var2=sub2_for_set1, var3=sub3_for_set1, ... }
    { var1=sub1_for_set2, var2=sub2_for_set2, var3=sub3_for_set2, ... }
    { var1=sub1_for_set3, var2=sub2_for_set3, var3=sub3_for_set3, ... }
}
```

or:

```
file name.template {
```

```

pattern { var1, var2, var3, ... }
  { sub1_for_set1, sub2_for_set1, sub3_for_set1, ... }
  { sub1_for_set2, sub2_for_set2, sub3_for_set2, ... }
  { sub1_for_set3, sub2_for_set3, sub3_for_set3, ... }
}

```

The first line (file name.template) specifies the record instance input file. The file name may appear inside double quotation marks; these are required if the name contains any characters that are not in the following set, or if it contains environment variable macros of the form `${ENV_VAR_NAME}` which are to be expanded before the file is opened:

```
a-z A-Z 0-9 _ + - . / \ : ; [ ] < >
```

Each set of definitions enclosed in `{ }` is variable substitution for the input file. The input file has each set applied to it to produce one composite file with all the completed substitutions in it. Version 1 should be obvious. In version 2, the variables are listed in the “pattern{ }” line, which must precede the braced substitution lines. The braced substitution lines contains sets which match up with the pattern{ } line.

6.20.1 EXAMPLE

Two simple template file examples are shown below. The examples specify the same substitutions to perform: `this=sub1` and `that=sub2` for a first set, and `this=sub3` and `that=sub4` for a second set.

```

file test.template {
  { this=sub1,that=sub2 }
  { this=sub3,that=sub4 }
}

```

```

file test.template {
  pattern{this,that}
  {sub1,sub2}
  {sub3,sub4 }
}

```

Assume that the file test.template contains:

```

record(ai,"$(this)record") {
  field(DESC,"this = $(this)")
}
record(ai,"$(that)record") {
  field(DESC,"this = $(that)")
}

```

Using dbLoadTemplate with either input is the same as defining the records:

```

record(ai,"sub1record") {
  field(DESC,"this = sub1")
}
record(ai,"sub2record") {
  field(DESC,"this = sub2")
}

record(ai,"sub3record") {
  field(DESC,"this = sub3")
}
record(ai,"sub4record") {

```

```
    field(DESC,"this = sub4")  
}
```

6.21 *dbReadTest*

```
dbReadTest -Idir -Smacsub file.dbd ... file.db ...
```

This utility can be used to check for correct syntax in database definition and database instance files. It just reads all the specified files

Multiple `-I`, and `-S` options can be specified. An arbitrary number of database definition and database instance files can be specified.

Chapter 7: IOC Initialization

7.1 Overview - Environments requiring a main program

If a main program is required (most likely on all environments except vxWorks and RTEMS), then initialization is performed by statements residing in startup scripts passed to iocsh. An example main program is:

```
int main(int argc, char *argv[])
{
    if(argc>=2) {
        iocsh(argv[1]);
        epicsThreadSleep(.2);
    }
    iocsh(NULL);
    return(0);
}
```

The first call to `iocsh` executes the commands from the filename passed as an argument to the program containing `main`. The second call to `iocsh` puts `iocsh` into interactive mode. This allows the user to issue the commands described in chapter "IOC Test Facilities" as well as some commands like `show` and `help`.

The file passed as the argument to the command contains statements like:

```
dbLoadDatabase("../db/<appname>.dbd",0,0)
registerRecordDeviceDriver(pdbbase)
dbLoadRecords("../db/<file>.db")
iocInit()
```

7.2 Overview - vxWorks

After vxWorks is loaded at IOC boot time, commands like the following, normally in the vxWorks startup command file, are issued to load and initialize the control system software:

```
# For many board support packages the following must be added
#cd <full path to target bin directory>
< cdCommands
cd topbin
ld < <appname>.munch

#The following uses drvTS for vxWorks.
#May be needed for hardware event systems
#TSinit

cd top
dbLoadDatabase("dbd/<appname>.dbd")
```

```
registerRecordDeviceDriver(pdbbase)
dbLoadRecords("db/<file>.db")
. . .
cd startup
iocInit()
```

`cdCommands` defines `vxWorks` global variables that allow `vxWorks` `cd` commands for convenient locations. For example in one of my test areas the following `cdCommands` file appears:

```
startup = "/home/phoebus6/MRK/epics/example/R3-14/iocBoot/iocexample"
appbin = "/home/phoebus6/MRK/epics/example/R3-14/bin/vxWorks-68040"
top = "/home/phoebus6/MRK/epics/example/R3-14"
topbin = "/home/phoebus6/MRK/epics/example/R3-14/bin/vxWorks-68040"
```

NOTE: This file is automatically generated via make rules.

The `ld` command loads EPICS core, record/device/driver support, and application specific modules.

`dbLoadDatabase` loads database definition files describing the record/device/driver support used by the application..

`dbLoadRecords` loads record instance definitions.

`iocInit` initializes the various epics components.

7.3 Overview - RTEMS

RTEMS applications can start up in many different ways depending on the board-support package for a particular piece of hardware. Systems which use the `Cexp` package can be treated much like `vxWorks`. Other systems first read initialization parameters from non-volatile memory or from a BOOTP/DHCP server. The exact mechanism depends upon the BSP. TFTP or NFS filesystems are then mounted and the IOC shell is used to read commands from a startup script. The location of this startup script is specified by a initialization parameter. This script can often be very similar, or identical to the one used with `vxWorks`. The RTEMS startup code calls

```
int epicsRtemsInitPreSetBootConfigFromNVRAM(struct rtems_bsdnet_config *config);
```

just before setting the initialization parameters from non-volatile memory and calls

```
int epicsRtemsInitPreSetBootConfigFromNVRAM(struct rtems_bsdnet_config *config);
```

just after setting the initialization parameters. An application can provide one or both of these routines to perform any custom initialization. These function prototypes and some useful external variable declarations are located in

```
epicsRtemsInitHooks.h
```

7.4 *iocInit*

`iocInit` performs the following functions:

7.4.1 `coreRelease`

Prints a messages showing which version of `iocCore` is being loaded.

7.4.2 taskwdInit

start the task watchdog task. This task accepts requests to watch other tasks. It runs periodically and checks to see if any of the tasks is suspended. If so it issues an error message. It can also optionally invoke a callback routine

7.4.3 callbackInit

Start the general purpose callback tasks. Three tasks are started with the only difference being scheduling priority.

7.4.4 dbCaLinkInit

Calls `dbCaLinkInit`. The initializes the task that handles database channel access links.

7.4.5 initDrvSup

`InitDrvSup` locates each device driver entry table and calls the init routine of each driver.

7.4.6 initRecSup

`InitRecSup` locates each record support entry table and calls the init routine.

7.4.7 initDevSup

`InitDevSup` locates each device support entry table and calls the init routine with an argument specifying that this is the initial call.

7.4.8 initDatabase

`InitDatabase` makes three passes over the database performing the following functions:

- Pass 1: Initializes following fields: `rset`, `dset`, `mlis`. Calls record support `init_record` (First pass)
- Pass 2: Convert each `PV_LINK` to `DB_LINK` or `CA_LINK`
- Pass 3: Calls record support `init_record` (second pass)

7.4.9 dbLockInitRecords

`dbLockInitRecords` creates the lock sets.

7.4.10 finishDevSup

`InitDevSup` locates each device support entry table and calls the init routine with an argument specifying that this is the finish call.

7.4.11 scanInit

The periodic, event, and io event scanners are initialized and started.

7.4.12 asInit

asInit initializes access security.

7.4.13 dbPutNotifyInit

dbPutNotifyInit initializes support for put notifies.

7.4.14 initialProcess

initialProcess processes all records that have PINI set true.

7.4.15 interruptAccept

A global variable "interruptAccept" is set TRUE. Until this time all interrupts should be ignored.

7.4.16 rsrv_init

The Channel Access servers are started

7.5 *Changing iocCore fixed limits*

The following commands can be issued after iocCore is loaded to change iocCore fixed limits. The commands should be given before any dbLoad commands.

```
callbackSetQueueSize(size)
dbPvdTableSize(size)
scanOnceSetQueueSize(size)
errlogInit(bufferSize)
```

7.5.1 callbackSetQueueSize

Requests for the general purpose callback tasks are placed in a ring buffer. This command can be used to set the size for the ring buffers. The default is 2000. A message is issued when a ring buffer overflows. It should rarely be necessary to override this default. Normally the ring buffer overflow messages appear when a callback task fails.

7.5.2 dbPvdTableSize

Record instance names are stored in a process variable directory, which is a hash table. The default number of hash entries is 512. dbPvdTableSize can be called to change the size. It must be called before any dbLoad commands and must be a power of 2 between 256 and 65536. If an IOC contains very large databases (several thousand) then a larger hash table size speeds up searches for records.

7.5.3 scanOnceSetQueueSize

scanOnce requests are placed in a ring buffer. This command can be used to set the size for the ring buffer. The default is 1000. t should rarely be necessary to override this default. Normally the ring buffer overflow messages appear when the scanOnce task fails.

7.5.4 errlogInit

Thus overrides the default buffer size for the errlog message queue. The default is 1280 bytes.

7.6 TSconfigure

NOTES:

- This is no longer supplied with base. It is available as an unbundled product.
- The aps version is supported on vxWorks and on RTEMS
- If TSconfigure is called then TSinit must also be called before iocInit

EPICS supports several methods for an IOC to obtain time so that accurate time stamps can be generated. The default is to obtain NTP time stamps from another computer. The following can be used to change the defaults. If an argument is given the value 0 then the default is applied.

TSconfigure(master, sync_rate, clock_rate, master_port, slave_port)

- **master:** 1=master timing IOC, 0=slave timing, default is slave.
- **sync_rate:** The clock sync rate in seconds. This rate tells how often the synchronous time stamp support software will confirm that an IOC clock is synchronized. The default is 10 seconds.
- **clock_rate:** The frequency in hertz of the clock, the default is 1000Hz for the event system. The value will be set to the IOC's internal clock rate when soft timing is used.
- **master_port:** UDP port for master. The default is 18233
- **slave_port:** UDP port for slave.
- **time_out:** UDP information request time out in milliseconds, if zero is entered here, the default will be used which is 250ms.
- **type:** 0=normal operation, 1=force soft timing type

See "Synchronous Time Stamp Support", by Jim Kowalkowski for details. Note that the default is to be a slave. If no master is found the slave will obtain a starting time from Unix.

7.7 initHooks

NOTE: starting with release 3.13.0beta12 initHooks was changed drastically (thanks to Benjamin Franksen at BESY). Old initHooks.c functions will still work but users are encouraged to switch to the new method.

The inithooks facility allows application specific functions to be called at various states during ioc initialization. The states are defined in initHooks.h, which contains the following definitions:

```
typedef enum {
    initHookAtBeginning,
    initHookAfterCallbackInit,
```

```
    initHookAfterCaLinkInit,  
    initHookAfterInitDrvSup,  
    initHookAfterInitRecSup,  
    initHookAfterInitDevSup,  
    initHookAfterInitDatabase,  
    initHookAfterFinishDevSup,  
    initHookAfterScanInit,  
    initHookAfterInitialProcess,  
    initHookAfterInterruptAccept,  
    initHookAtEnd  
}initHookState;  
  
typedef void (*initHookFunction)(initHookState state);  
int initHookRegister(initHookFunction func);
```

Any new functions that are registered before `iocInit` reaches the desired state will be called when `iocInit` reaches that state. The following is skeleton code to use the facility:

```
static initHookFunction myHookFunction;  
  
int myHookInit(void)  
{  
    return(initHookRegister(myHookFunction));  
}  
  
static void myHookFunction(initHookState state)  
{  
    switch(state) {  
        case initHookAfterInitRecSup:  
            ...  
            break;  
        case initHookAfterInterruptAccept:  
            ...  
            break;  
        default:  
            break;  
    }  
}
```

An arbitrary number of functions can be registered.

7.8 Environment Variables

The following environment variables are used by `iocCore`:

```
EPICS_CA_ADDR_LIST  
EPICS_CA_CONN_TMO  
EPICS_CA_BEACON_PERIOD  
EPICS_CA_AUTO_ADDR_LIST  
EPICS_CA_REPEATER_PORT  
EPICS_CA_SERVER_PORT  
EPICS_CA_MAX_ARRAY_BYTES
```

```
EPICS_TS_NTP_INET  
EPICS_IOC_LOG_PORT  
EPICS_IOC_LOG_INET
```

For an explanation of the EPICS_CA_... variables see the EPICS Channel Access Reference Manual. For an explanation of the EPICS_IOC_LOG_... variables see IOC Error Logging: iocLogClient in this manual. EPICS_TS_NTP_INET is used only on vxWorks. It gives the location of the Network Time Protocol server. If it is not defined the IOC uses the boot server as its NTP server.

These variables can be overridden via the epicsEnvSet function. For example:

```
epicsEnvSet("EPICS_CA_CONN_TMO","10")
```

All epicsEnvSet commands should be issued after iocCore is loaded and before any dbLoad commands.

The following commands can be issued to iocsh:

epicsPrtEnvParams - This shows all environment variables used by iocCore.

epicsEnvShow - This shows all environment variables on your system.

7.9 Initialize Logging

Initialize the logging system. See chapter "IOC Error Logging" for details. For now just realise that the following can be used if you want to use a private host log file.

```
epicsEnvSet("EPICS_IOC_LOG_PORT", "<port>")  
epicsEnvSet("EPICS_IOC_LOG_INET", "<inet addr>")
```

These command must be given immediately after iocCore is loaded.

To start logging you must issue the command:

```
iocLogInit
```


Chapter 8: Access Security

8.1 Overview

This chapter describes access security. i.e. the system that limits access to IOC databases. It consists of the following sections:

1. Overview - This section
2. Quick start - A summary of the steps necessary to start access security.
3. User's Guide - This explains what access security is and how to use it.
4. Design Summary - Functional Requirements and Design Overview.
5. Application Programmer's Interface
6. Database Access Security - Access Security features for EPICS IOC databases.
7. Channel Access Security - Access Security features in Channel Access
8. Trapping Channel Access Writes - This allows trapping of all writes from external channel access clients.
9. Implementation Overview

The requirements for access security were generated at ANL/APS in 1992. The requirements document is:

EPICS: Channel Access Security - Functional Requirements, Ned D. Arnold, 03/-9/92.

This document is available through the EPICS website.

8.2 Quick Start

In order to "turn on" access security for a particular IOC the following must be done:

- Create the access security file.
- IOC databases may have to be modified
 - Record instances may have to have values assigned to field ASG. If ASG is null the record is in group DEFAULT.
 - Access security files can be reloaded after iocInit via a subroutine record with `asSubInit` and `asSubProcess` as the associated subroutines. Writing the value 1 to this record will cause a reload.
- The vxWorks startup file must contain the following command before iocInit.
`asSetFilename("accessSecurityFile")`
The following is an optional command.
`asSetSubstitutions("var1=sub1,var2=sub2,...")`

The following rules decide if access security is turned on for an IOC:

- If `asSetFilename` is not executed before `iocInit`, access security will NEVER be started..
- If `asSetFile` is given and any error occurs while first initializing access security, then ALL access to that ioc is denied.

- If after successfully starting access security, an attempt is made to restart and an error occurs then the previous access security configuration is maintained.

After an IOC has been booted with access security enabled, the access security rules can be changed by issuing the `asSetFilename`, `asSetSubstitutions`, and `asInit`. The functions `asInitialize`, `asInitFile`, and `asInitFP`, which are described below, can also be used.

8.3 User's Guide

8.3.1 Features

Access security protects IOC databases from unauthorized Channel Access Clients. Access security is based on the following:

- **Who:** Userid of the channel access client.
- **Where:** Hostid where the user is logged on. This is the host on which the channel access client exists. Thus no attempt is made to see if a user is local or is remotely logged on to the host.
- **What:** Individual fields of records are protected. Each record has a field containing the Access Security Group (ASG) to which the record belongs. Each field has an access security level, which must be 0 or 1. The security level is defined in the ascii record definition file. Thus the access security level for a field is the same for all record instances of a record type.
- **When:** Access rules can contain input links and calculations similar to the calculation record.

8.3.2 Limitations

An IOC database can be accessed only via Channel Access or via the `vxWorks` or `ioc` shell. It is assumed that access to the local IOC console is protected via physical security and `telnet/rlogin` access protected via normal networking and physical security methods.

No attempt has been made to protect against the sophisticated saboteur. Network security methods must be used to limit access to the subnet on which the iocs reside.

8.3.3 Definitions

This document uses the following terms:

- **ASL:** Access Security Level (Called access level in Req Doc)
- **ASG:** Access Security Group (Called PV Group in Req Doc)
- **UAG:** User Access Group
- **HAG:** Host Access Group

8.3.4 Access Security Configuration File

This section describes the format of a file containing definitions of the user access groups, host access groups, and access security groups. An IOC creates an access configuration database by reading an access configuration file (the extension `.acf` is recommended). Lets first give a simple example and then a complete description of the syntax.

8.3.4.1 Simple Example

```
UAG(uag) {user1,user2}
HAG(hag) {host1,host2}
ASG(DEFAULT) {
  RULE(1,READ)
  RULE(1,WRITE) {
    UAG(uag)
    HAG(hag)
  }
}
```

These rules provide read access to anyone located anywhere and write access to user1 and user2 if they are located at host1 or host2.

8.3.4.2 Syntax Definition

In the following description:

- [] Surrounds optional elements
- | Separates alternatives
- ... Means that an arbitrary number of definitions may be given.

Any line beginning with a # character is a comment

The elements <name>, <user>, <host>, <pvname>, and <calculation> can be given a quoted or unquoted strings. The rules for unquoted strings are the same as for database definitions.

```
UAG(<name>) [{ <user> [, <user> ...] }]
...
HAG(<name>) [{ <host> [, <host> ...] }]
...
ASG(<name>) [{
  [ INP<index>(<pvname>)
  ...]
  RULE(<level>,NONE | READ | WRITE [, NOTRAPWRITE | TRAPWRITE]) {
    [UAG(<name> [, <name> ...])]
    [HAG(<name> [, <name> ...])]
    CALC(<calculation>)
  }
}
...
}]
...
```

8.3.4.3 Discussion

- **UAG:** User Access Group. This is a list of user names. The list may be empty. The same user can appear in multiple UAGs. For vxWorks iocs the user name is taken from the user field of the boot parameters.
- **HAG:** Host Access Group. This is a list of host names. It may be empty. The same host name can appear in multiple HAGs. For iocs the host name is taken from the target name of the boot parameters. NOTE: host names are converted to lower case.
- **ASG:** An access security group. The group "DEFAULT" is a special case. If a member specifies a null group or a group which has no ASG definition then the member is assigned to the group "DEFAULT".

- **INP<index>** Index must have one of the values “A” to “L”. These are just like the INP fields of a calculation record. It is necessary to define INP fields if a CALC field is defined in any RULE for the ASG.
- **RULE** This defines access permissions. <level> must be 0 or 1. Permission for a level 1 field implies permission for level 0 fields. The permissions are NONE, READ, and WRITE. WRITE permission implies READ permission. The standard EPICS record types have all fields set to level 1 except for VAL, CMD (command), and RES (reset). An optional argument specifies if writes should be trapped. See the section below on trapping Channel Access writes for how this is used. If not given the default is NOTRAPWRITE.
 - **UAG** specifies a list of user access groups that can have the access privilege. If UAG is not defined then all users are allowed.
 - **HAG** specifies a list of host access groups that have the access privilege. If HAG is not defined then all hosts are allowed.
 - **CALC** is just like the CALC field of a calculation record except that the result must evaluate to TRUE or FALSE. The rule only applies if the calculation result is TRUE, where the actual test for TRUE is $(0.99 < result < 1.01)$. Anything else is regarded as FALSE and will cause the rule to be ignored. Assignment statements are not permitted in CALC expressions here.

Each IOC record contains a field ASG, which specifies the name of the ASG to which the record belongs. If this field is null or specifies a group which is not defined in the access security file then the record is placed in group "DEFAULT".

The access privilege for a channel access client is determined as follows:

1. The ASG associated with the record is searched.
2. Each RULE is checked for the following:
 - a. The field's level must be less than or equal to the level for this RULE.
 - b. If UAG is defined, the user must belong to one of the specified UAGs. If UAG is not defined all users are accepted.
 - c. If HAG is defined, the user's host must belong to one one of the HAGs. If HAG is not defined all hosts are accepted.
 - d. If CALC is specified, the calculation must yield the value 1, i.e. TRUE. If any of the INP fields associated with this calculation are in INVALID alarm severity the calculation is considered false. The actual test for TRUE is $.99 < result < 1.01$.
3. The maximum access allowed by step 2 is the access chosen.

Multiple RULEs can be defined for a given ASG, even RULEs with identical levels and access permission.

8.3.5 ascheck - Check Syntax of Access Configuration File

After creating or modifying an access configuration file it can be checked for syntax errors by issuing the command:

```
ascheck -S "xxx=yyy,..." < "filename"
```

This is a Unix command. It displays errors on stdout. If no errors are detected it prints nothing. Only syntax errors not logic errors are detected. Thus it is still possible to get your self in trouble. The flag -S means a set of macro substitutions may appear. This is just like the macro substitutions for dbLoadDatabase.

8.3.6 IOC Access Security Initialization

In order to have access security turned on during IOC initialization the following command must appear in the startup file before iocInit is called:

```
asSetFilename("<access security file>")
```

If this command does not appear then access security will not be started by iocInit. If an error occurs when iocInit calls asInit then all access to the ioc is disabled, i.e. no channel access client will be able to access the ioc.

Access security also supports macro substitution just like `dbLoadDatabase`. The following command specifies the desired substitutions:

```
asSetSubstitutions("var1=sub1,var2=sub2,...")
```

This command must be issued before `iocInit`.

After an IOC is initialized the access security database can be changed. The preferred way is via the subroutine record described in the next section. It can also be changed by issuing the following command to the `vxWorks` shell:

```
asInit
```

It is also possible to reissue `asSetFilename` and/or `asSetSubstitutions` before `asInit`. If any error occurs during `asInit` the old access security configuration is maintained. It is **NOT** permissible to call `asInit` before `iocInit` is called.

Restarting access security after `ioc` initialization is an expensive operation and should not be used as a regular procedure.

8.3.7 Database Configuration

8.3.7.1 Access Security Group

Each database record has a field `ASG` which holds a character string. Any database configuration tool can be used to give a value to this field. If the `ASG` of a record is not defined or is not equal to a `ASG` in the configuration file then the record is placed in `DEFAULT`.

8.3.7.2 Subroutine Record Support

Two subroutines, which can be attached to a subroutine record, are available (provided with `iocCore`):

```
asSubInit
asSubProcess
```

NOTE: These subroutines are automatically registered thus do **NOT** put a `registrar` definition in your database definition file.

If a record is created that attaches to these routines, it can be used to force the IOC to load a new access configuration database. To change the access configuration:

1. Modify the file specified by the last call to `asSetFilename` so that it contains the new configuration desired.
2. Write a 1 to the subroutine record `VAL` field. Note that this can be done via channel access.

The following action is taken:

3. When the value is found to be 1, `asInit` is called and the value set back to 0.
4. The record is treated as an asynchronous record. Completion occurs when the new access configuration has been initialized or a time-out occurs. If initialization fails the record is placed into alarm with a severity determined by `BRSV`.

8.3.7.3 Record Type Description

Each field of each record type has an associated access security level of `ASL0` or `ASL1`. See the chapter "Database Definition" for details.

8.3.8 Example:

Lets design a set of rules for a Linac. Assume the following:

1. Anyone can have read access to all fields at anytime.
2. Linac engineers, located in the injection control or control room, can have write access to most level 0 fields only if the Linac is not in operational mode.
3. Operators, located in the injection control or control room, can have write access to most level 0 fields anytime.
4. The operations supervisor, linac supervisor, and the application developers can have write access to all fields but must have some way of not changing something inadvertently.
5. Most records use the above rules but a few (high voltage power supplies, etc.) are placed under tighter control. These will follow rules 1 and 4 but not 2 or 3.
6. IOC channel access clients always have level 1 write privilege.

Most Linac IOC records will not have the ASG field defined and will thus be placed in ASG "DEFAULT". The following records will have an ASG defined:

- LI:OPSTATE and any other records that need tighter control have ASG="critical". One such record could be a subroutine record used to cause a new access configuration file to be loaded. LI_OPSTATE has the value (0,1) if the Linac is (not operational, operational).
- LI:levlpermit has ASG="permit". In order for the opSup, linacSup, or an appDev to have write privilege to everything this record must be set to the value 1.

The following access configuration satisfies the above rules.

```
UAG(op) {op1,op2,superguy}
UAG(opSup) {superguy}
UAG(linac) {waw,nassiri,grelick,berg,fuja,gsm}
UAG(linacSup) {gsm}
UAG(appDev) {nda,kko}
HAG(icr) {silver,phebos,gaea}
HAG(cr) {mars,hera,gold}
HAG(ioc) {ioclic1,ioclic2,ioclid1,ioclid2,ioclid3,ioclid4,ioclid5}
ASG(DEFAULT) {
  INPA(LI:OPSTATE)
  INPB(LI:levlpermit)
  RULE(0,WRITE) {
    UAG(op)
    HAG(icr,cr)
    CALC("A=1")
  }
  RULE(0,WRITE) {
    UAG(op,linac,appdev)
    HAG(icr,cr)
    CALC("A=0")
  }
  RULE(1,WRITE) {
    UAG(opSup,linacSup,appdev)
    CALC("B=1")
  }
  RULE(1,READ)
  RULE(1,WRITE) {
    HAG(ioc)
  }
}
```

```
}
ASG(permit) {
  RULE(0,WRITE) {
    UAG(opSup,linacSup,appDev)
  }
  RULE(1,READ)
  RULE(1,WRITE) {
    HAG(ioc)
  }
}
ASG(critical) {
  INPB(LI:lvlpermit)
  RULE(1,WRITE) {
    UAG(opSup,linacSup,appdev)
    CALC("B=1")
  }
  RULE(1,READ)
  RULE(1,WRITE) {
    HAG(ioc)
  }
}
```

8.4 Design Summary

8.4.1 Summary of Functional Requirements

A brief summary of the Functional Requirements is:

1. Each field of each record type is assigned an access security level.
2. Each record instance is assigned to a unique access security group.
3. Each user is assigned to one or more user access groups.
4. Each node is assigned to a host access group.
5. For each access security group a set of access rules can be defined. Each rule specifies:
 - a. Access security level
 - b. READ or READ/WRITE access.
 - c. An optional list of User Access Groups or * meaning anyone.
 - d. An optional list of Host Access Groups or * meaning anywhere.
 - e. Conditions based on values of process variables

8.4.2 Additional Requirements

8.4.2.1 Performance

Although the functional requirements doesn't mention it, a fundamental goal is performance. The design provides almost no overhead during normal database access and moderate overhead for the following: channel access client/server connection, ioc initialization, a change in value of a process variable referenced by an access calculation, and dynamically

changing a records access control group. Dynamically changing the user access groups, host access groups, or the rules, however, can be a time consuming operation. This is done, however, by a low priority IOC task and thus does not impact normal ioc operation.

8.4.2.2 Generic Implementation

Access security should be implemented as a stand alone system, i.e. it should not be imbedded tightly in database or channel access.

8.4.2.3 No Access Security within an IOC

Within an IOC no access security is invoked. This means that database links and local channel access clients calls are not subject to access control. Also test routines such as dbgf should not be subject to access control.

8.4.2.4 Defaults

It must be possible to easily define default access rules.

8.4.2.5 Access Security is Optional

When an IOC is initialized, access security is optional.

8.4.3 Design Overview

The implementation provides a library of routines for accessing the security system. This library has no knowledge of channel access or IOC databases, i.e. it is generic. Database access, which is responsible for protecting an IOC database, calls library routines to add each IOC record to one of the access control groups.

Lets briefly discuss the access security system and how database access and channel access interact with it.

8.4.3.1 Configuration File

User access groups, host access groups, and access security groups are configured via an ASCII file.

8.4.3.2 Access Security Library

The access security library consists of the following groups of routines: initialization, group manipulation, client manipulation, access computation, and diagnostic. The initialization routine reads a configuration file and creates a memory resident access control database. The group manipulation routines allow members to be added and removed from access groups. The client routines provide services for clients attached to members.

8.4.3.3 IOC Database Access Security

The interface between an IOC database and the access security system.

8.4.3.4 Channel Access Security

Whenever the Channel Access broadcast server receives a `ca_search` request and finds the process variable, it calls `asAddClient`. Whenever it disconnects it calls `asRemoveClient`. Whenever it issues a `get` or `put` to the database it must call `asCheckGet` or `asCheckPut`.

8.4.4 Comments

It is likely that the access rules will be defined such that many IOCs will attach to a common process variable. As a result the IOC containing the PV will have many CA clients.

What about password protection and encryption? I maintain that this is a problem to be solved in a level above the access security described in this document. This is the issue of protecting against the sophisticated saboteur.

8.4.5 Performance and Memory Requirements

Performance has not yet been measured but during the tests to measure memory usage no noticeable change in performance during ioc initialization or during Channel Access clients connection was noticed. Unless access privilege is violated the overhead during channel access gets and puts is only an extra comparison.

In order to measure memory usage, the following test was performed:

1. A database consisting of 5000 soft analog records was created.
2. A channel access client (`caput`) was created that performs `ca_puts` on each of the 5000 channels. Each time it begins a new set of puts the value increments by 1.
3. A channel access client (`caget`) was created that has monitors on each of the 5000 channels.

The memory consumption was measured before `iocInit`, after `iocInit`, after `caput` connected to all channels, and after `caget` connected to all 5000 channels. This was done for APS release 3.11.5 (before access security) and the first version which included access security. The results were:

	R3.11.5	After
Before iocInit	4,244,520	4,860,840
After iocInit	4,995,416	5,964,904
After caput	5,449,780	6,658,868
After caget	8,372,444	9,751,796

Before the database was loaded the memory used was 1,249,692 bytes. Thus most of the memory usage before `iocInit` resulted from storage for records. The increase since R3.11.5 results from added fields to `dbCommon`. Fields were added for access security, synchronous time support and for the new caching put support. The other increases in memory usage result from the control blocks needed to support access control. The entire design was based on maximum performance. This resulted in increased memory usage.

8.5 Access Security Application Programmer's Interface

8.5.1 Introduction

File `asLib.h` describes the access security data structures and the last section of this chapter has a diagram describing the relationship between the structures. The structures are:

- ASBASE - Contains the list head for lists of UAGs, HAGs, and ASGs
- UAG - A user access group.
- HAG - A host access group
- ASG - An access security group. It contains the list head for ASGINPs, ASGRULEs, and ASGMEMBERS
- ASGINP - Contains the information for an INPx.
- ASGRULE - Contains the information for a rule

- ASGMEMBER - Contains the information for a member of an access security group. It contains the list head for ASGCLIENTs.

All structures except ASGMEMBER and ASGCLIENT are created by the access security library itself when it reads an access security file. An ASGMEMBER is created each time asAddMember is called by code that interfaces to the database. An ASGCLIENT is created each time asAddClient is called by a channel access server.

8.5.2 Definitions

The following are descriptions of arguments of routines described later.

```
typedef struct asgMember *ASMEMBERPVT;
typedef struct asgClient *ASCLIENTPVT;
typedef int (*ASINPUTFUNCPTR)(char *buf,int max_size);
typedef enum{
    asClientCOAR /*Change of access rights*/
    /*For now this is all*/
} asClientStatus;
typedef void (*ASCLIENTCALLBACK)(ASCLIENTPVT,asClientStatus);
```

8.5.3 Initialization

```
long asInitialize(ASINPUTFUNCPTR inputFunction)
long asInitFile(const char *filename,const char *substitutions)
long asInitFP(FILE *fp,const char *substitutions)
```

These routines read an access definition file and perform all initialization necessary. The caller must provide a routine to provide input lines for asInitialize. asInitFile and asInitFP do their own input and also perform macro substitutions.

The initialization routines can be called multiple times. If an access system already exists the old definitions are removed and the new one initialized. Existing members are placed in the new ASGs.

8.5.4 Group manipulation

The routines are called by code that knows how to associate ASG names with the database. In the case of IOC databases, dbCommon has a field ASG. At IOC initialization a call is made to asAddMember for every record instance in the IOC database.

8.5.4.1 add Member

```
long asAddMember(ASMEMBERPVT *ppv, const char *asgName);
```

This routine adds a new member to ASG asgName. The calling routine must provide storage for ASMEMBERPVT. Upon successful return *ppv will be equal to the address of storage used by the access control system. The access system keeps an orphan list for all asgNames not defined in the access configuration.

The caller must provide permanent storage for asgName.

This routine returns S_asLib_asNotActive without doing anything if access control is not active.

8.5.4.2 remove Member

```
long asRemoveMember(ASMEMBERPVT *ppv);
```

This routine removes a member from an access control group. If any clients are still present it returns an error status of `S_asLib_clientExists` without removing the member.

This routine returns `S_asLib_asNotActive` without doing anything if access control is not active.

8.5.4.3 get Member Pvt

```
void *asGetMemberPvt(ASMEMBERPVT pvt);
```

For each member, the access system keeps a pointer that can be used by the caller. This routine returns the value of the pointer.

This routine returns `NULL` if access security is not active

8.5.4.4 put Member Pvt

```
long asPutMemberPvt(ASMEMBERPVT pvt, void *userPvt);
```

This routine is used to set the pointer returned by `asGetMemberPvt`.

This routine returns `S_asLib_asNotActive` without doing anything if access control is not active.

8.5.4.5 change Group

```
long asChangeGroup(ASMEMBERPVT *ppvt, const char *newAsgName);
```

This routine changes the group for an existing member. The access rights of all clients of the member are recomputed.

The caller must provide permanent storage for `newAsgName`.

This routine returns `S_asLib_asNotActive` without doing anything if access control is not active.

8.5.5 Client Manipulation

This code is called by a channel access server.

8.5.5.1 add Client

```
long asAddClient(ASCLIENTPVT *ppvt, ASMEMBERPVT pvt, int asl,
                const char *user, char *host);
```

This routine adds a client to an ASG member. The calling routine must provide storage for `ASCLIENTPVT`. `ASMEMBERPVT` is the value that was set by calling `asAddMember`. The database code and the server code must develop a convention that allows the server code to locate the `ASMEMBERPVT`. For IOC databases, `ASMEMBERPVT` is kept in `dbCommon`. `asl` is the access security level.

The caller must provide permanent storage for `user` and `host`. Note that `user` is "const char *" but `host` is just "char *". The reason is the host names are converted to lower case.

This routine returns `S_asLib_asNotActive` without doing anything if access control is not active.

8.5.5.2 change Client

```
long asChangeClient(ASCLIENTPVT ppvt, int asl,
                   const char *user, char *host);
```

This routine changes one or more of the values `asl`, `user`, and `host` for an existing client. Again the caller must provide permanent storage for `user` and `host`. It is permissible to use the same `user` and `host` used in the call to `asAddClient` with different values.

This routine returns `S_asLib_asNotActive` without doing anything if access control is not active.

8.5.5.3 remove Client

```
long asRemoveClient(ASCLIENTPVT *pvt);
```

This call removes a client.

This routine returns `S_asLib_asNotActive` without doing anything if access control is not active.

8.5.5.4 get Client Pvt

```
void *asGetClientPvt(ASCLIENTPVT pvt);
```

For each client, the access system keeps a pointer that can be used by the caller. This routine returns the value of the pointer.

This routine returns `NULL` if access security is not active.

8.5.5.5 put Client Pvt

```
void asPutClientPvt(ASCLIENTPVT pvt, void *userPvt);
```

This routine is used to set the pointer returned by `asGetClientPvt`.

8.5.5.6 register Callback

```
long asRegisterClientCallback(ASCLIENTPVT pvt,  
ASCLIENTCALLBACK pcallback);
```

This routine registers a callback that will be called whenever the access privilege of the client changes.

This routine returns `S_asLib_asNotActive` without doing anything if access control is not active.

8.5.5.7 check Get

```
long asCheckGet(ASCLIENTPVT pvt);
```

This routine, actually a macro, returns `(TRUE,FALSE)` if the client (has, doesn't have) get access rights.

8.5.5.8 check Put

```
long asCheckPut(ASCLIENTPVT pvt);
```

This routine, actually a macro, returns `(TRUE,FALSE)` if the client (has, doesn't have) put access rights.

8.5.5.9 asTrapWriteBefore and asTrapWriteAfter

```
void *asTrapWriteBefore(ASCLIENTPVT clientPvt,  
const char *userid, const char *hostid, void *serverSpecific);  
void *asTrapWriteAfter(void *trapPvt);
```

These routines must be called before and after any write performed for a client. The value returned by `asTrapWriteBefore` must be the value passed to `asTrapWriteAfter`. The `serverSpecific` argument is assigned to the `serverSpecific` field of the `asTrapWriteMessage` described below.

8.5.6 Access Computation

8.5.6.1 compute all Asg

```
long asComputeAllAsg(void);
```

This routine calls `asComputeAsg` for each access security group.

This routine returns `S_asLib_asNotActive` without doing anything if access control is not active.

8.5.6.2 compute Asg

```
long asComputeAsg(ASG *pasg);
```

This routine calculates all `CALC` entries for the ASG and calls `asCompute` for each client of each member of the specified access security group.

This routine returns `S_asLib_asNotActive` without doing anything if access control is not active.

8.5.6.3 compute access rights

```
long asCompute(ASCLIENTPVT pvt);
```

This routine computes the access rights of a client. This routine is normally called by the access library itself rather than use code.

This routine returns `S_asLib_asNotActive` without doing anything if access control is not active.

8.5.7 Diagnostics

8.5.7.1 Dump

```
int asDump(void (*member)(ASMEMBERPVT),  
           void (*client)(ASCLIENTPVT),int verbose);  
int asDumpFP(FILE *fp,void (*member)(ASMEMBERPVT),  
            void (*client)(ASCLIENTPVT),int verbose);
```

These routines print the current access security database. If `verbose` is 0 (`FALSE`), then only the information obtained from the access security file is printed.

If `verbose` is `TRUE` then additional information is printed. The value of each `INP` is displayed. The list of members belonging to each ASG and the clients belonging to each member are displayed. If member callback is specified as an argument, then it is called for each member. If client callback is specified, it is called for each access security client.

8.5.7.2 Dump UAG

```
int asDumpUag(char *uagname)  
int asDumpUagFP(FILE *fp,char *uagname)
```

These routines display the specified UAG or if `uagname` is `NULL` each UAG defined in the access security database.

8.5.7.3 Dump HAG

```
int asDumpHag(char *hagname)  
int asDumpHagFP(FILE *fp,char *hagname)
```

These routines display the specified UAG or if `uagname` is `NULL` each UAG defined in the access security database.

8.5.7.4 Dump Rules

```
int asDumpRules(char *asgname)  
int asDumpRulesFP(FILE *fp,char *asgname)
```

These routines display the rules for the specified ASG or if `asgname` is `NULL` the rules for each ASG defined in the access security database.

8.5.7.5 Dump member

```
int asDumpMem(char *asgname,
              void (*memcallback)(ASMEMBERPVT),int clients)
int asDumpMemFP(FILE *fp,char *asgname,
               void (*memcallback)(ASMEMBERPVT),int clients)
```

This routine displays the member and, if `clients` is `TRUE`, client information for the specified ASG or if `asgname` is `NULL` the member and client information for each ASG defined in the access security database. It also calls `memcallback` for each member if this argument is not `NULL`.

8.5.7.6 Dump hash table

```
int asDumpHash(void)
int asDumpHash(FILE *fp,void)
```

These show the contents of the hash table used to locate UAGs and HAGs,

8.6 Database Access Security

8.6.1 Access Level definition

The definition of access level means that a level is defined for each field of each record type.

1. Structure `fldDes` (`dbBase.h`), which describes the attributes of each field, contains a field `access_security_level`. In addition definitions exist for the symbols: `ASL0` and `ASL1`.
2. Each field description in a record description contains a field with the value `ASLx`.

The meanings of the Access Security Level definitions are as follows:

- `ASL0` Assigned to fields used during normal operation
- `ASL1` Assigned to fields that may be sensitive to change. Permission to access this level implies permission for `ASL0`.

Most record types assign `ASL` as follows: The fields `VAL`, `RES` (Reset), and `CMD` use the value `ASL0`. All other fields use `ASL1`.

8.6.2 Access Security Group definition

`dbCommon` contains the fields `ASG` and `ASP`. `ASG` (Access Security Group) is a character string. The value can be assigned via a database configuration tool or else a utility could be provided to assign values during `ioc` initialization. `ASP` is an access security private field. It contains the address of an `ASGMEMBER`.

8.6.3 Access Client Definition

Struct `dbAddr` contains a field `asPvt`, which contains the address of an `ASGCLIENT`. This definition is also added to struct `db_addr` so that old database access also supports access security.

8.6.4 Database Access Library

Two files `asDbLib.c` and `asCa.c` implement the interface between IOC databases and access control. It contains the following routines:

8.6.4.1 Initialization

```
int asSetFilename(char *acf)
```

Calling this routine sets the filename of an access configuration file. The next call to `asInit` uses this file. This routine must be called before `iocInit` otherwise access configuration is disabled. If access security is disabled during `iocInit` it will never be turned on.

```
int asSetSubstitutions(char *substitutions)
```

This routine specifies macro substitutions.

```
int asInit()
int asInitAsyn(ASDBC_CALLBACK *pcallback)
```

This routines call `asInitialize`. If the current access configuration file, as specified by `asSetFilename`, is `NULL` then the routine just returns, otherwise the configuration file is used to create the access configuration database.

This routine is called by `iocInit`. `asInit` can also be called at any time to change the access configuration information.

`asInitAsyn` spawns a task `asInitTask` to perform the initialization. This allows `asInitAsyn` to be called from a subroutine called by the process entry of a subroutine record. `asInitTask` calls `taskwdInsert` so that if it suspends for some reason `taskwd` can detect the failure. After initialization all records in the database are made members of the appropriate access control group.

If the caller provides an `ASDBC_CALLBACK` then when either initialization completes or `taskwd` detects a failure the users callback routine is called via one of the standard callback tasks.

`asInitAsyn` will return a value of `-1` if access initialization is already active. It returns `0` if `asInitTask` is successfully spawned.

8.6.4.2 Routines used by Channel Access Server

```
int asDbGetAsl(void *paddr)
```

Get Access Security level for the field referenced by a database access structure. The argument is defined as a `void*` so that both old and new database access can be used.

```
void * asDbGetMemberPvt(void *paddr)
```

Get `ASMEMBERPVT` for the field referenced by a database access structure. The argument is defined as a `void*` so that both old and new database access can be used.

8.6.4.3 Routine to test asAddClient

```
int astac(char *pname, char *user, char *host)
```

This is a routine to test `asAddClient`. It simulates the calls that are made by Channel Access.

8.6.4.4 Subroutines attached to a subroutine record

These routines are provided so that a channel access client can force an `ioc` to load a new access configuration database.

```
long asSubInit(struct subRecord *prec,int pass)
long asSubProcess(struct subRecord *prec)
```

These are routines that can be attached to a subroutine record. Whenever a 1 is written to the record, `asSubProcess` calls `asInit`. If `asInit` returns success, it returns with asynchronously. When `asInitTask` calls the completion routine supplied by `asSubProcess`, the return status is used to place the record in alarm.

8.6.4.5 Diagnostic Routines

These routines provide interfaces to the `asDump` routines described in the previous chapter. They do NOT lock before calling the associated routine. Thus they may fail if the access security configuration is changing while they are running. However the danger of the user accidentally aborting a command and leaving the access security system locked is considered a risk that should be avoided.

```
asdbdump(void)
asdbdumpFP(FILE *fp)
```

These routines call `asDumpFP` with a member callback and with verbose `TRUE`.

```
aspuag(char *uagname)
aspuagFP(FILE *fp,char *uagname)
```

These routines call `asDumpUagFP`.

```
asphag(char *hagname)
asphagFP(FILE *fp,char *hagname)
```

These routines call `asDumpHagFP`.

```
asprules(char *asgname)
asprulesFP(FILE *fp,char *asgname)
```

These routines call `asDumpRulesFP`.

```
aspmem(char *asgname,int clients)
aspmemFP(FILE *fp,char *asgname,int clients)
```

These routines call `asDumpMemFP`.

8.7 Channel Access Security

EPICS Access Security was originally designed to protect Input Output Controllers (IOCs) from unauthorized access via the Channel Access (CA) network protocol. It can also be used by any Channel Access Server (CAS) tool. For example the Channel Access PV Gateway implements its own access security. This section describes the interaction between a CA server and the Access Security system. It also briefly describes how the current access rights state is communicated to clients of the EPICS control system via the CA client interface.

8.7.1 CA Server Interfaces to the Access Security System

The CA server calls `asAddClient()` and `asRegisterClientCallback()` for each of the channels that a client connects to the server. The routine `asRemoveClient()` is called whenever the client clears (removes) a channel or when the client disconnects.

The server maintains storage for the clients host and user names. The initial value of these strings are supplied to the server when the client connects and can be updated at any time by the client. When these strings change then `asChangeClient()` is called for each of the channels maintained by the server for the client.

The server checks for read access when processing gets and for write access when processing puts. If access is denied then an exception message is sent to the client. The macros `asCheckGet()` and `asCheckPut()` perform the checks.

The server checks for read access when processing requests to register an event callback (monitor) for the client. If there is read access the server always sends an initial update indicating the current value. If there isn't read access the server sends one update indicating no read access and disables subsequent updates.

The server receives asynchronous notification of access rights change via the callback registered with `asRegisterClientCallback()`. When a channel's access rights change the server communicates the current state to the client library. If read access to a channel is lost and there are events (monitors) registered on the channel then the server sends an update to the client for each of them indicating no access and disables future updates for each event. If read access is reestablished to a channel and there are events (monitors) registered on the channel then the server re-enables updates and sends an initial update message to the client for each of them.

The server must also call `asTrapWriteBefore()` and `asTrapWriteAfter()` before and after a put request from a client is performed.

8.7.2 Client Interfaces

Additional details on the channel access client side callable interfaces to access security can be obtained from the "Channel Access Reference Manual".

The client library stores and maintains the current state of the access rights for each channel that it has established. The client library receives asynchronous updates of the current access rights state from the server. It uses this state to check for read access when processing gets and for write access when processing puts. If a program issues a channel access request that is inconsistent with the client library's current knowledge of the access rights state then access is denied and an error code is returned to the application. The current access rights state as known by the client library can be tested by an applications program with the C macros `ca_read_access()` and `ca_write_access()`.

An application program can also receive asynchronous notification of changes to the access rights state by registering a function to be called back when the client library updates its storage of the access rights state. The application's call back function is installed for this purpose by calling `ca_replace_access_rights_event()`.

If the access rights state changes in the server after a request is queued in the client library but before the request is processed by the server then it is possible that the request will fail in the server. Under these circumstances then an exception will be raised in the client.

The server always sends one update to the client when the event (monitor) is initially registered. If there isn't read access then the status in the arguments to the application program's event call back function indicates no read access and the value in the arguments to the clients event call back is set to zero. If the read access right changes after the event is initially registered then another update is supplied to the application programs call back function.

8.8 Trapping Channel Access Writes

Access security provides a facility `asTrapWrite` that can trap write requests and pass them to any facility that registers a listener. In order to use this facility three things are necessary:

1. The facility, e.g. the channel access server, using access security must make two calls: `asTrapWriteBefore()` and `asTrapWriteAfter()`. These are defined in `asLib.h`. The channel access server on the ioc makes these calls.
2. `asTrapWrite()` gets called by `asTrapWriteBefore()` and `asTrapWriteAfter()` and uses the `TRAPWRITE` option specified with the RULEs given in the access configuration file to decide if listeners should be called. `asTrapWrite` also includes a routine `asTrapWriteRegisterListener()`.
3. Some facility not included with access security must call `asTrapWriteRegisterListener()`. If nothing calls `asTrapWriteRegisterListener`, `asTrapWrite` does nothing.

The remainder of this section describes how a facility can use `asTrapWrite.h`, which is defined as:

```
typedef struct asTrapWriteMessage {
    const char *userid;
    const char *hostid;
    void *serverSpecific;
    void *userPvt;
} asTrapWriteMessage;

typedef void *asTrapWriteId;
typedef void(*asTrapWriteListener)(asTrapWriteMessage *pmessage,int after);

asTrapWriteId asTrapWriteRegisterListener(asTrapWriteListener func);
void asTrapWriteUnregisterListener(asTrapWriteId id);
```

After a facility calls `asTrapWriteRegisterListener()` its `asTrapWriteListener()` will get called before and after each write with an associated RULE that has the option `TRAPWRITE` set.

`asTrapWriteRegisterListener()` is passed the address of an `asTrapWriteMessage`. This message contains the following fields:

- `userid` - Userid of whoever originated the request.
- `hostid` - Hostid of whoever originated the request.
- `serverSpecific` - The meaning of this field is server specific. If the listener uses this field it must know what type of server is supplying the messages. It is the value the server provides to `asTrapWriteBefore`.
- `userPvt` - This field is for use by the `asTrapWriteListener`. When the listener is called before the write, `userPvt` has the value 0. The listener can give it any value it desires and `userPvt` will have the same value when the listener gets called after the write.

`asTrapWriteListener` delays the associated server thread so it must not do anything that causes it to block.

The IOC's RSRV server the calls `asTrapWriteBefore` with `serverSpecific` set to the `dbAddr` describing the database location.

8.9 Access Control: Implementation Overview

This section provides a few aids for reading the access security code. Include file `asLib.h` describes the control blocks used by the access security library.

8.9.1 Implementation Overview

The following files form the access security system:

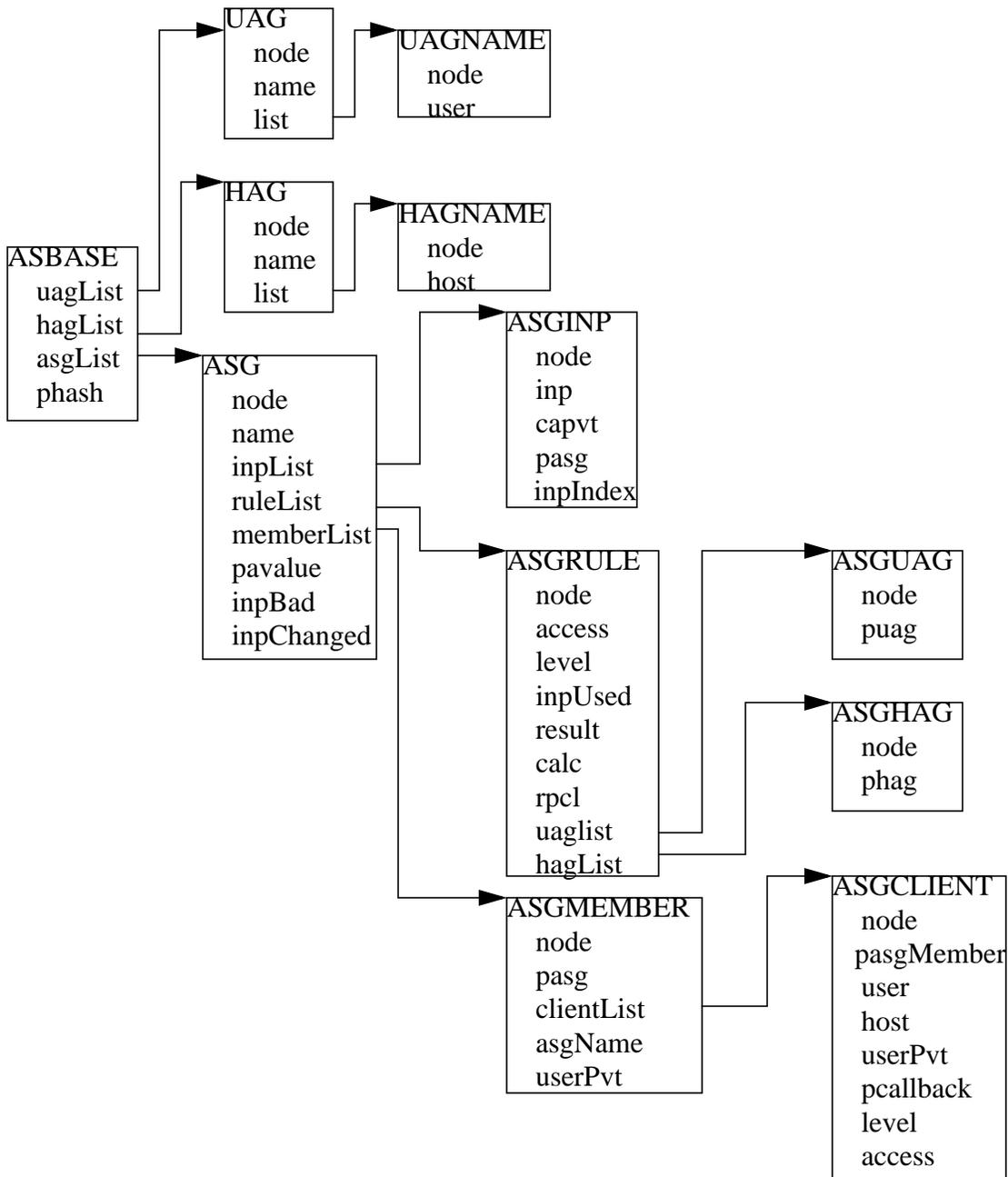
- **asLib.h** Definitions for the portion of access security that is independent of IOC databases.
- **asDbLib.h** Definitions for access routines that interface to an IOC database.
- **asLib_lex.l** `Lex` and `Yacc` (actually EPICS `flex` and `antelope`) are used to parse the access configuration file. This is the `lex` input file.
- **asLib.y** This is the `yacc` input file. Note that it includes `asLibRoutines.c`, which do most of the work.
- **asLibRoutines.c** These are the routines that implement access security. This code has no knowledge of the database or channel access. It is a general purpose access security implementation.
- **asDbLib.c** This contains the code for interfacing access security to the IOC database.
- **asCa.c** This code contains the channel access client code that implements the `INP` and `CALC` definitions in an access security database.
- **ascheck.c** The Unix program which performs a syntax check on a configuration file.

8.9.2 Locking

Because it is possible for multiple tasks to simultaneously modify the access security database it is necessary to provide locking. Rather than try to provide low level locking, the entire access security database is locked during critical operations. The only things this should hold up are access initialization, CA searches, CA clears, and diagnostic routines. It should NEVER cause record processing to wait. In addition CA gets and puts should never be delayed. One exception exists. If the ASG field of a record is changed then `asChangeGroup` is called which locks.

All operations invoked from outside the access security library that cause changes to the internal structures of the access security database.routines lock.

8.10 Structures



Chapter 9: IOC Test Facilities

9.1 Overview

This chapter describes a number of IOC test routines that are of interest to both application developers and system developers. The routines are available via `iocsh` or the `vxWorks` shell. For both shells, the parentheses are optional, but the arguments must be separated by commas. On `vxWorks` all character string arguments must be enclosed in `""`. For `iocsh` the `""` are optional. For example:

```
dbpf("aiTest","2")
dbpf "aiTest","2"
```

are both valid with both `iocsh` and with the `vxWorks` shell.

```
dbpf aiTest 2
```

Is valid for `iocsh` but not for the `vxWorks` shell.

`iosch` also allows output redirection, i.e. the output of any `iosch` command can be redirected to a file. For example

```
dbl > dbl.lst
```

Will put the output of the `dbl` command in file `"dbl.lst"`.

The user should also be aware of the field `TPRO`, which is present in every database record. If it is set `TRUE` then a message is printed each time its record is processed and a message is printed for each record processed as a result of it being processed.

If `iocsh` is being used it provides help for all commands that have been registered. Just type

```
help
```

or

```
help <command>
```

9.2 Database List, Get, Put

9.2.1 dbl

Database List:

```
dbl("<record type>","<field list>")
```

Examples

```
dbl
dbl("ai")
```

```
dbl( "*" )  
dbl( " " )
```

This command prints the names of records in the run time database. If `<record type>` is empty(" "), "*" , or not specified, all records are listed. If `<record type>` is specified, then only the names of the records of that type are listed.

If `<field list>` is given and not empty then the values of the fields specified are also printed.

9.2.2 dbgrep

List Record Names That Match a Pattern:

```
dbgrep( "<pattern>" )
```

Examples

```
dbgrep( "S0*" )  
dbgrep( "*gpibAi*" )
```

Lists all record names that match a pattern. The pattern can contain any characters that are legal in record names as well as "*", which matches 0 or more characters.

9.2.3 dba

Database Address:

```
dba( "<record_name.field_name>" )
```

Example

```
dba( "aitest" )  
dba( "aitest.VAL" )
```

This command calls `dbNameToAddr` and then prints the value of each field in the `dbAddr` structure describing the field. If the field name is not specified then VAL is assumed (the two examples above are equivalent).

9.2.4 dbgf

Get Field:

```
dbgf( "<record_name.field_name>" )
```

Example:

```
dbgf( "aitest" )  
dbgf( "aitest.VAL" )
```

This performs a `dbNameToAddr` and then a `dbGetField`. It prints the field type and value. If the field name is not specified then VAL is assumed (the two examples above are equivalent). Note that `dbGetField` locks the record lockset, so `dbgf` will not work on a record with a stuck lockset; use `dbpr` instead in this case.

9.2.5 dbpf

Put Field:

```
dbpf( "<record_name.field_name>", "<value>" )
```

Example:

```
dbpf("aitest","5.0")
```

This command performs a `dbNameToAddr` followed by a `dbPutField` and `dbgf`. If `<field_name>` is not specified VAL is assumed.

9.2.6 dbpr

Print Record:

```
dbpr("<record_name>",<interest level>)
```

Example

```
dbpr("aitest",2)
```

This command prints all fields of the specified record up to and including those with the indicated interest level. Interest level has one of the following values:

- **0:** Fields of interest to an Application developer and that can be changed as a result of record processing.
- **1:** Fields of interest to an Application developer and that do not change during record processing.
- **2:** Fields of major interest to a System developer.
- **3:** Fields of minor interest to a System developer.
- **4:** Fields of no interest.

9.2.7 dbtr

Test Record:

```
dbtr("<record_name>")
```

This calls `dbNameToAddr`, then `dbProcess` and finally `dbpr` (interest level 3). Its purpose is to test record processing.

9.2.8 dbnr

Print number of records:

```
dbnr(<all_recordtypes>)
```

This command displays the number of records of each type and the total number of records. If `all_record_types` is 0 then only record types with record instances are displayed otherwise all record types are displayed.

9.3 Breakpoints

A breakpoint facility that allows the user to step through database processing on a per lockset basis. This facility has been constructed in such a way that the execution of all locksets other than ones with breakpoints will not be interrupted. This was done by executing the records in the context of a separate task.

The breakpoint facility records all attempts to process records in a lockset containing breakpoints. A record that is processed through external means, e.g.: a scan task, is called an entrypoint into that lockset. The `dbstat` command described below will list all detected entrypoints to a lockset, and at what rate they have been detected.

9.3.1 dbb

Set Breakpoint:

```
dbb( "<record_name>" )
```

Sets a breakpoint in a record. Automatically spawns the `bkptCont`, or breakpoint continuation task (one per lockset). Further record execution in this lockset is run within this task's context. This task will automatically quit if two conditions are met, all breakpoints have been removed from records within the lockset, and all breakpoints within the lockset have been continued.

9.3.2 dbd

Remove Breakpoint:

```
dbd( "<record_name>" )
```

Removes a breakpoint from a record.

9.3.3 dbs

Single Step:

```
dbs( "<record_name>" )
```

Steps through execution of records within a lockset. If this command is called without an argument, it will automatically step starting with the last detected breakpoint.

9.3.4 dbc

Continue:

```
dbc( "<record_name>" )
```

Continues execution until another breakpoint is found. This command may also be called without an argument.

9.3.5 dbp

Print Fields Of Suspended Record:

```
dbp( "<record_name>,<interest_level>" )
```

Prints out the fields of the last record whose execution was suspended.

9.3.6 dbap

Auto Print:

```
dbap( "<record_name>" )
```

Toggles the automatic record printing feature. If this feature is enabled for a given record, it will automatically be printed after the record is processed.

9.3.7 dbstat

Status:

```
dbstat
```

Prints out the status of all locksets that are suspended or contain breakpoints. This lists all the records with breakpoints set, what records have the autoprint feature set (by dbap), and what entrypoints have been detected. It also displays the vxWorks task ID of the breakpoint continuation task for the lockset. Here is an example output from this call:

```
LSet: 00009   Stopped at: so#B: 00001   T: 0x23cafac
             Entrypoint: so#C: 00001   C/S:      0.1
             Breakpoint: so(ap)
LSet: 00008#B: 00001   T: 0x22fee4c
             Breakpoint: output
```

The above indicates that two locksets contain breakpoints. One lockset is stopped at record “so.” The other is not currently stopped, but contains a breakpoint at record “output.” “LSet:” is the lockset number that is being considered. “#B:” is the number of breakpoints set in records within that lockset. “T:” is the vxWorks task ID of the continuation task. “C:” is the total number of calls to the entrypoint that have been detected. “C/S:” is the number of those calls that have been detected per second. (ap) indicates that the autoprint feature has been turned on for record “so.”

9.4 Error Logging

9.4.1 eltc

Display error log messages on console:

```
eltc(int noYes)
```

This determines if error messages are displayed on the IOC console. 0 means no and any other value means yes.

9.5 Hardware Reports

9.5.1 dbior

I/O Report:

```
dbior (<driver_name>,<interest level>)
```

This command calls the report entry of the indicated driver. If <driver_name> is "" or "*", then a report for all drivers is generated. The command also calls the report entry of all device support modules. Interest level is one of the following:

- **0:** Print a short report for each module.
- **1:** Print additional information.
- **2:** Print even more info. The user may be prompted for options.

9.5.2 dbhcr

Hardware Configuration Report:

```
dbhcr()
```

This command produces a report of all hardware links. To use it on the IOC, issue the command:

```
dbhcr > report
```

The report will probably not be in the sort order desired. The Unix command:

```
sort report > report.sort
```

should produce the sort order you desire.

9.6 Scan Reports

9.6.1 scanppl

Print Periodic Lists:

```
scanppl(double rate)
```

This routine prints a list of all records in the periodic scan list of the specified rate. If rate is 0.0 all period lists are shown.

9.6.2 scanpel

Print Event Lists:

```
scanpel(int event_number)
```

This routine prints a list of all records in the event scan list for the specified event number. If event_number is 0 all event scan lists are shown.

9.6.3 scanpiol

Print I/O Event Lists:

```
scanpiol
```

This routine prints a list of all records in the I/O event scan lists.

9.7 Time Server Report

NOTE: TSreport is implemented by drvTS.c. It is only available on vxWorks

9.7.1 TSreport

Format:

TSreport

This routine prints out information about the Time server. This includes:

- Slave or Master
- Soft or Hardware synchronized
- Clock and Sync rates
- etc.

9.8 Access Security Commands

9.8.1 asSetSubstitutions

Format:

```
asSetSubstitutions("substitutions")
```

Specifies macro substitutions used when access security is initialized.

9.8.2 asSetFilename

Format:

```
asSetFilename("<filename>")
```

This command defines a new access security file.

9.8.3 asInit

Format:

```
asInit
```

This command reinitializes the access security system. It rereads the access security file in order to create the new access security database. This command is useful either because the `asSetFilename` command was used to change the file or because the file itself was modified. Note that it is also possible to reinitialize the access security via a subroutine record. See the access security document for details.

9.8.4 asdbdump

Format:

```
asdbdump
```

This provides a complete dump of the access security database.

9.8.5 aspuag

Format:

```
aspuag("<user access group>")
```

Print the members of the user access group. If no user access group is specified then the members of all user access groups are displayed.

9.8.6 asphag

Format:

```
asphag("<host access group>")
```

Print the members of the host access group. If no host access group is specified then the members of all host access groups are displayed.

9.8.7 asprules

Format:

```
asprules("<access security group>")
```

Print the rules for the specified access security group or if no group is specified for all groups.

9.8.8 aspmem

Format:

```
aspmem("<access security group>", <print clients>)
```

Print the members (records) that belong to the specified access security group, for all groups if no group is specified. If <print clients> is (0, 1) then Channel Access clients attached to each member (are not, are) shown.

9.9 Channel Access Reports

9.9.1 casr

Channel Access Server Report

```
casr(<level>)
```

Level can have one of the following values:

0

Prints server's protocol version level and a one line summary for each client attached. The summary lines contain the client's login name, client's host name, client's protocol version number, and the number of channel created within the server by the client.

1

Level one provides all information in level 0 and adds the task id used by the server for each client, the client's IP protocol type, the file number used by the server for the client, the number of seconds elapsed since the last request was received from the client, the number of seconds elapsed since the last response was sent to the client, the number of unprocessed request bytes from the client, the number of response bytes which have not been flushed to the client, the client's IP address, the client's port number, and the client's state.

2

Level two provides all information in levels 0 and 1 and adds the number of bytes allocated by each client and a list of channel names used by each client. Level 2 also provides information about the number of bytes in the server's free memory pool, the distribution of entries in the server's resource hash table, and the list of IP addresses to which the server is sending beacons. The channel names are shown in the form:

<name>(nrw)

where

n is number of ca_add_events the client has on this channel

r is (-,R) if client (does not, does) have read access to the channel.

w is (-, W) if client (does not, does) have write access to the channel.

9.9.2 dbel

Format:

```
dbel ("<record_name>")
```

This routine prints the Channel Access event list for the specified record.

9.9.3 dbcar

Database to Channel Access Report - See "Record Link Reports"

9.9.4 ascar

Format:

```
ascar(level)
```

Prints a report of the channel access links for the INP fields of the access security rules. Level 0 produces a summary report. Level 1 produces a summary report plus details on any unconnect channels. Level 2 produces the summary nreport plus a detail report on each channel.

9.10 *Interrupt Vectors*

9.10.1 veclist

Format:

```
veclist
```

NOTE: This routine is only available on vxWorks. On PowerPC CPUs it requires BSP support to work, and even then it cannot display chained interrupts using the same vector.

Print Interrupt Vector List

9.11 Miscellaneous

9.11.1 epicsParamShow

Format:

```
    epicsParamShow  
or  
    epicsPrtEnvParams
```

Print the environment variables that are created with epicsEnvSet. These are defined in <base>/config/CONFIG_ENV and <base>/config/CONFIG_SITE_ENV or else by user applications calling epicsEnvSet.

9.11.2 epicsEnvShow

Format:

```
    epicsEnvShow( "<name>" )
```

Show Environment variables. On vxWorks it shows the variables created via calls to putenv.

9.11.3 epicsRelease

Format:

```
    coreRelease
```

Print release of iocCore.

9.12 Database System Test Routines

These routines are normally only of interest to EPICS system developers NOT to Application Developers.

9.12.1 dbtgf

Test Get Field:

```
    dbtgf( "<record_name.field_name>" )
```

Example:

```
    dbtgf( "aitest" )  
    dbtgf( "aitest.VAL" )
```

This performs a dbNameToAddr and then calls dbGetField with all possible request types and options. It prints the results of each call. This routine is of most interest to system developers for testing database access.

9.12.2 dbtpf

Test Put Field:

```
dbtpf("<record_name.field_name>", "<value>")
```

Example:

```
dbtpf("aitest", "5.0")
```

This command performs a `dbNameToAddr`, then calls `dbPutField`, followed by `dbgF` for each possible request type. This routine is of interest to system developers for testing database access.

9.12.3 dbtpn

Test Put Notify:

```
dbtpn("<record_name.field_name>", "<value>")
```

Example:

```
dbtpn("aitest", "5.0")
```

This command performs a `dbNameToAddr`, then calls `dbPutNotify` and has a callback routine that prints a message when it is called. This routine is of interest to system developers for testing database access.

9.13 Record Link Reports

9.13.1 dblsr

Lock Set Report:

```
dblsrc(<recordname>, <level>)
```

This command generates a report showing the lock set to which each record belongs. If `recordname` is 0, "", or "*" all records are shown, otherwise only records in the same lock set as `recordname` are shown.

`level` can have the following values:

- 0 - Show lock set information only.
- 1 - Show each record in the lock set.
- 2 - Show each record and all database links in the lock set.

9.13.2 dbcar

Database to channel access report

```
dbcar(<recordname>, <level>)
```

This command generates a report showing database channel access links. If `recordname` is "*" then information about all records is shown otherwise only information about the specified record.

`level` can have the following values:

- 0 - Show summary information only.
- 1 - Show summary and each CA link that is not connected.
- 2 - Show summary and status of each CA link.

9.13.3 dbhcr

Report hardware links. See “Hardware Reports”.

9.14 Old Database Access Testing

These routines are of interest to EPICS system developers. They are used to test the old database access interface, which is still used by Channel Access.

9.14.1 gft

Get Field Test:

```
gft("<record_name.field_name>")
```

Example:

```
gft("aitest")  
gft("aitest.VAL")
```

This performs a `db_name_to_addr` and then calls `db_get_field` with all possible request types. It prints the results of each call. This routine is of interest to system developers for testing database access.

9.14.2 pft

Put Field Test:

```
pft("<record_name.field_name>","<value>")
```

Example:

```
pft("aitest","5.0")
```

This command performs a `db_name_to_addr`, `db_put_field`, `db_get_field` and prints the result for each possible request type. This routine is of interest to system developers for testing database access.

9.14.3 tpn

Test Put Notify:

```
tpn("<record_name.field_name>","<value>")
```

Example:

```
tpn("aitest","5.0")
```

This routine tests `dbPutNotify` via the old database access interface.

9.15 Routines to dump database information

9.15.1 dbDumpPath

Dump Path:

```
dbDumpPath(pdbbase)
```

```
dbDumpPath(pdbbase)
```

The current path for database includes is displayed.

9.15.2 dbDumpMenu

Dump Menu:

```
dbDumpMenu(pdbbase, "<menu>")
```

```
dbDumpMenu(pdbbase, "menuScan")
```

If the second argument is 0 then all menus are displayed.

9.15.3 dbDumpRecordType

Dump Record Description:

```
dbDumpRecordType(pdbbase, "<record type>")
```

```
dbDumpRecordType(pdbbase, "ai")
```

If the second argument is 0 then all descriptions of all records are displayed.

9.15.4 dbDumpField

Dump Field Description:

```
dbDumpField(pdbbase, "<record type>", "<field name>")
```

```
dbDumpField(pdbbase, "ai", "VAL")
```

If the second argument is 0 then the field descriptions of all records are displayed. If the third argument is 0 then the description of all fields are displayed.

9.15.5 dbDumpDevice

Dump Device Support:

```
dbDumpDevice(pdbbase, "<record type>")
```

```
dbDumpDevice(pdbbase, "ai")
```

If the second argument is 0 then the device support for all record types is displayed.

9.15.6 dbDumpDriver

Dump Driver Support:

```
dbDumpDriver(pdbbase)
```

```
dbDumpDriver(pdbbase)
```

9.15.7 dbDumpRecord

Dump Record Instances:

```
dbDumpRecord(pdbbase, "<record type>", level)
```

```
dbDumpRecords(pdbbase, "ai")
```

If the second argument is 0 then the record instances for all record types is displayed. The third argument determines which fields are displayed just like for the command `dbpr`.

9.15.8 dbDumpBreaktable

Dump breakpoint table

```
dbDumpBreaktable(pdbbase, name)
```

```
dbDumpBreaktable(pdbbase, "typeKdegF")
```

This command dumps a breakpoint table. If the second argument is 0 all breakpoint tables are dumped.

9.15.9 dbPvdDump

Dump the Process variable Directory:

```
dbPvdDump(pdbbase, verbose)
```

```
dbPvdDump(pdbbase, 0)
```

This command shows how many records are mapped to each hash table entry of the process variable directory. If verbose is not 0 then the command also displays the names which hash to each hash table entry.

Chapter 10: IOC Error Logging

10.1 Overview

Errors detected by an IOC can be divided into classes: Errors related to a particular client and errors not attributable to a particular client. An example of the first type of error is an illegal Channel Access request. For this type of error, a status value should be passed back to the client. An example of the second type of error is a device driver detecting a hardware error. This type of error should be reported to a system wide error handler.

Dividing errors into these two classes is complicated by a number of factors.

- In many cases it is not possible for the routine detecting an error to decide which type of error occurred.
- Normally, only the routine detecting the error knows how to generate a fully descriptive error message. Thus, if a routine decides that the error belongs to a particular client and merely returns an error status value, the ability to generate a fully descriptive error message is lost.
- If a routine always generates fully descriptive error messages then a particular client could cause error message storms.
- While developing a new application the programmer normally prefers fully descriptive error messages. For a production system, however, the system wide error handler should not normally receive error messages cause by a particular client.

If used properly, the error handling facilities described in this chapter can process both types of errors.

This chapter describes the following:

- Error Message Generation Routines - Routines which pass messages to the `errlog` Task.
- Error Log Listeners - Any code can register to receive `errlog` messages.
- `errlogThread` - A thread that passes the messages to all registered listeners.
- console output and message buffer size - Messages can also be written to the console. The storage for the message queue can be specified by the user.
- status codes - EPICS status codes.
- `iocLog` - A system wide error logger supplied with base. It writes all messages to a system wide file.
NOTE: Many sites use `CMLOG` instead of `iocLog`.

NOTE: `recGbl` error routines are also provided. They in turn call one of the error message routines.

10.2 Error Message Routines

10.2.1 Basic Routines

```
int errlogPrintf(const char *pformat, ...);
int errlogVprintf(const char *pformat,va_list pvar);
int errlogMessage(const char *message);
```

```
void errlogFlush(void);
```

`errlogPrintf` and `errlogVprintf` are like `printf` and `vprintf` provided by the standard C library, except that the output is sent to the `errlog` task. Consult any book that describes the standard C library such as "The C Programming Language ANSI C Edition" by Kernighan and Ritchie. If `epicsThreadIsOkToBlock` is true, which is true during `iocInit`, `errlogVprintf` does NOT send output to the `errlog` task.

`errlogMessage` sends message to the `errlog` task.

`errlogFlush` wakes up the `errlog` task and then waits until all messages are flushed from the queue.

10.2.2 Log with Severity

```
typedef enum {
    errlogInfo, errlogMinor, errlogMajor, errlogFatal
}errlogSevEnum;

int errlogSevPrintf(const errlogSevEnum severity,
    const char *pformat, ...);
int errlogSevVprintf(const errlogSevEnum severity,
    const char *pformat, va_list pvar);

char *errlogGetSevEnumString(const errlogSevEnum severity);

void errlogSetSevToLog(const errlogSevEnum severity );
errlogSevEnum errlogGetSevToLog(void);
```

`errlogSevPrintf` and `errlogSevVprintf` are like `errlogPrintf` and `errlogVprintf` except that they add the severity to the beginning of the message in the form "sevr=<value>" where value is one of "info, minor, major, fatal". Also the message is suppressed if severity is less than the current severity to suppress. If `epicsThreadIsOkToBlock` is true, which is true during `iocInit`, `errlogSevVprintf` does NOT send output to the `errlog` task.

`errlogGetSevEnumString` gets the string value of severity.

`errlogSetSevToLog` sets the severity to log. `errlogGetSevToLog` gets the current severity to log.

10.2.3 Status Routines

```
void errMessage(long status, char *message);

void errPrintf(long status, const char *pFileName,
    int lineno, const char *pformat, ...);
```

Routine `errMessage` (actually a macro that calls `errPrintf`) has the following format:

```
void errMessage(long status, char *message);
```

Where `status` is defined as:

- **0**: Find latest vxWorks or Unix error.
- **-1**: Don't report status.
- Other: See "Return Status Values" above.

`errMessage`, via a call to `errPrintf`, prints the message, the status symbol and string values, and the name of the task which invoked `errMessage`. It also prints the name of the source file and the line number from which the call was issued.

The calling routine is expected to pass a descriptive message to this routine. Many subsystems provide routines built on top of `errMessage` which generate descriptive messages.

An IOC global variable `errVerbose`, defined as an external in `errMdef.h`, specifies verbose messages. If `errVerbose` is `TRUE` then `errMessage` should be called whenever an error is detected even if it is known that the error belongs to a specific client. If `errVerbose` is `FALSE` then `errMessage` should be called only for errors that are not caused by a specific client.

Routine `errPrintf` is normally called as follows:

```
errPrintf(status, __FILE__, __LINE__, "<fmt>", ...);
```

Where `status` is defined as:

- **0**: Find latest vxWorks or Unix error.
- **-1**: Don't report status.
- Other: See "Return Status Values", above.

`FILE` and `LINE` are defined as:

- `__FILE__` As shown or `NULL` if the file name and line number should not be printed.
- `__LINE__` As shown

The remaining arguments are just like the arguments to the C `printf` routine. `errVerbose` determines if the filename and line number are shown.

An EPICS status code can also be converted to a string. If the supplied status code isn't registered in the status code database then the raw status code number is converted into a string in the destination buffer.

```
#include "errMdef.h"
void errSymLookup(long status, char *pBuf, unsigned bufLength);
```

10.2.4 Obsolete Routines

```
int epicsPrintf(const char *pformat, ...);
int epicsVprintf(const char *pformat, va_list pvar);
```

These are macros that call `errlogPrintf` and `errlogVprintf`. They are provided for compatibility.

10.3 *errlog* Listeners

Any code can receive `errlog` message. The following are the calls to add and remove a listener.

```
typedef void(*errlogListener)(void *pvt, const char *message);
void errlogAddListener(errlogListener listener, void *pPrivate);
void errlogRemoveListener(errlogListener listener);
```

These routines add/remove a callback that receives each error message. These routines are the interface to the actual system wide error handlers.

10.4 *errlogThread*

The error message routines can be called by any non-interrupt level code. These routines pass the message to the errlog Thread. If any of the error message routines are called at interrupt level, `epicsInterruptContextMessage` is called with the message "errlogPrintf called from interrupt level".

errlogThread manages the messages. Messages are placed in a message queue, which is read by errlogThread. The message queue uses a fixed block of memory to hold all messages. When the message queue is full additional messages are rejected but a count of missed messages is kept. The next time the message queue empties an extra message about the missed messages is generated.

The maximum message size is by default 256 characters. If a message is longer, the message is truncated and a message explaining that it was truncated is appended. There is a chance that long messages corrupt memory. This only happens if client code is defective. Long messages most likely result from "%s" formats with a bad string argument.

errlogThread passes each message to any registered listener.

10.5 *console output and message queue size*

The errlog system can also display messages on the ioc console. It calls `epicsThreadIsOkToBlock` to decide when to display the message. If it is OK to block, the message is displayed by the same thread that calls one of the errlog print routines. If it is not OK to block, `errlogThread` displays the messages.

Normally the errlog system displays all messages on the console. `eltc` can be used to suppress these messages.

```
int eltc(int yesno); /* error log to console (0 or 1) */
int errlogInit(int bufsize);
int errlogInit2(int bufsize, int maxMsgSize);
```

eltc determines if errlog task writes message to the console. During error messages storms this command can be used to suppress console messages. A argument of 0 suppresses the messages and any other value lets the message go to the console.

errlogInit or errlogInit2 can be used to initialize the error logging system with a larger buffer and maximum message size. The default buffer size is 1280 bytes, and the default maximum message size is 256.

10.6 *Status Codes*

EPICS defined status values provide the following features:

- Whenever possible, IOC routines return a status value: (0, non-0) means (OK, ERROR).
- The include files for each IOC subsystem contain macros defining error status symbols and strings.
- Routines are provided for run time access of the error status symbols and strings.
- A global variable `errVerbose` helps code decide if error messages should be generated.

WARNING: During the fall of 1995 a series of tech-talk messages were generated concerning EPICS status values. No consensus was reached.

Whenever it makes sense, IOC routines return a status value encoded similar to the vxWorks error status encoding. The most significant short word indicates the subsystem module within which the error occurred. The low order short word is a subsystem status value. In order that status values do not conflict with the vxWorks error status values all subsystem numbers are greater than 500.

A file `epics/share/epicsH/errMdef.h` defines each subsystem number. For example the `define` for the database access routines is:

```
#define M_dbAccess (501 << 16) \
    /*Database Access Routines*/
```

Directory `epics/share/epicsH` contains an include library for every IOC subsystem that returns standard status values. The status values are encoded with lines of the following format:

```
#define S_xxxxxxx value /*string value*/
```

For example:

```
#define S_dbAccessBadDBR (M_dbAccess|3) \
    /*Invalid Database Request*/
```

For example, when `dbGetField` detects a bad database request type, it executes the statement:

```
return(S_dbAccessBadDBR);
```

The calling routine checks the return status as follows:

```
status = dbGetField( ...);
if(status) { /* Call was not successful */ }
```

10.7 iocLog

NOTE: Many sites use CMLOG instead of iocLog. See the CMLOG documentation for details.

This consists of two modules: `iocLogServer` and `iocLogClient`. The client code runs on each ioc and listens for the messages generated locally by the `errlog` system. It also reports the messages from the `vxWorks logMsg` facility.

10.7.1 iocLogServer

This runs on a host. It receives messages for all enabled `iocLogClients` in the local area network. The messages are written to a file. Epics base provides a startup file `"base/src/util/rc2.logServer"`, which is a SystemV init script to start the server. Consult this script for details.

To start a log server on a UNIX or PC workstation you must first set the following environment variables and then run the executable `"iocLogServer"` on your PC or UNIX workstation.

EPICS_IOC_LOG_FILE_NAME

The name and path to the log file.

EPICS_IOC_LOG_FILE_LIMIT

The maximum size in characters for the log file (after which it becomes a circular file and writes new messages over old messages at the beginning of the file). If the value is zero then there is no limit on the size of the log file.

EPICS_IOC_LOG_FILE_COMMAND

A shell command string used to obtain the log file path name during initialization and in response to `SIGHUP`. The new path name will replace any path name supplied in `EPICS_IOC_LOG_FILE_NAME`.

Thus, if `EPICS_IOC_LOG_FILE_NAME` is

`"a/b/c.log"` and `EPICS_IOC_LOG_FILE_COMMAND` returns `"A/B"` or `"A/B/"` the log server will be stored at `"A/B/c.log"`

If `EPICS_IOC_LOG_FILE_COMMAND` is empty then this behavior is disabled. This feature is used at some sites for switching the server to a new directory at a fixed time each day. This variable is currently used only by the UNIX version of the log server.

`EPICS_IOC_LOG_PORT`

THE TCP/IP port used by the log server.

To configure an IOC so that its messages are placed in the log you must set the environment variable `EPICS_IOC_LOG_INET` to the IP address of the host that is running the log server, and `EPICS_IOC_LOG_PORT` to the TCP/IP port used by the log server.

Defaults for all of the above parameters are specified in the files `$(EPICS_BASE)/config/CONFIG_SITE_ENV` and `$(EPICS_BASE)/config/CONFIG_ENV`.

10.7.2 iocLogClient

This runs on each ioc. It is started by calling:

```
iocLogInit();
```

The global variable `iocLogDisable` can be used to enable/disable the messages from being sent to the server. Setting this variable to (0,1) (enables,disables) the messages generation. If `iocLogDisable` is set to 1 before calling `iocLogInit` then `iocLogClient` will not even initialize itself. `iocLogDisable` can also be changed to turn logging on or off.

`iocLogClient` calls `errlogAddListener` and sends each message to the `iocLogServer`.

10.7.3 Configuring a Private Log Server

In a testing environment it is desirable to use a private log server. This can be done as follows:

- Add a `epicsEnvSet` command to your IOC startup file. For example

```
ld < iocCore
epicsEnvSet("EPICS_IOC_LOG_INET=xxx.xxx.xxx.xxx")
```

The inet address is that of your host workstation.
- On your host workstation, start the log server.

Chapter 11: Record Support

11.1 Overview

The purpose of this chapter is to describe record support in sufficient detail such that a C programmer can write new record support modules. Before attempting to write new support modules, you should carefully study a few of the existing support modules. If an existing support module is similar to the desired module most of the work will already be done.

From previous chapters, it should be clear that many things happen as a result of record processing. The details of what happens are dependent on the record type. In order to allow new record types and new device types without impacting the core IOC system, the concept of record support and device support has been created. For each record type, a record support module exists. It is responsible for all record specific details. In order to allow a record support module to be independent of device specific details, the concept of device support has been created.

A record support module consists of a standard set of routines which are called by database access routines. These routines implement record specific code. Each record type can define a standard set of device support routines specific to that record type.

By far the most important record support routine is `process`, which `dbProcess` calls when it wants to process a record. This routine is responsible for the details of record processing. In many cases it calls a device support I/O routine. The next section gives an overview of what must be done in order to process a record. Next is a description of the entry tables that must be provided by record and device support modules. The remaining sections give example record and device support modules and describe some global routines useful to record support modules.

The record and device support modules are the only modules that are allowed to include the record specific include files as defined in `base/rec`. Thus they are the only routines that access record specific fields without going through database access.

11.2 Overview of Record Processing

The most important record support routine is `process`. This routine determines what record processing means. Before the record specific “`process`” routine is called, the following has already been done:

- Decision to process a record.
- Check that record is not active, i.e. `pact` must be `FALSE`.
- Check that the record is not disabled.

The `process` routine, together with its associated device support, is responsible for the following tasks:

- Set record active while it is being processed
- Perform I/O (with aid of device support)
- Check for record specific alarm conditions
- Raise database monitors
- Request processing of forward links

A complication of record processing is that some devices are intrinsically asynchronous. It is **NEVER** permissible to wait for a slow device to complete. Asynchronous records perform the following steps:

1. Initiate the I/O operation and set `pact TRUE`
2. Determine a method for again calling process when the operation completes
3. Return immediately without completing record processing
4. When process is called after the I/O operation complete record processing
5. Set `pact FALSE` and return

The examples given below show how this can be done.

11.3 Record Support and Device Support Entry Tables

Each record type has an associated set of record support routines. These routines are located via the data structures defined in `epics/share/epicsH/recSup.h`. The concept of record support routines isolates the `iocCore` software from the details of each record type. Thus new records can be defined and supported without affecting the IOC core software.

Each record type also has zero or more sets of device support routines. Record types without associated hardware, e.g. calculation records, normally do not have any associated device support. Record types with associated hardware normally have a device support module for each device type. The concept of device support isolates IOC core software and even record support from device specific details.

Corresponding to each record type is a set of record support routines. The set of routines is the same for every record type. These routines are located via a Record Support Entry Table (RSET), which has the following structure

```
rset { /* record support entry table */
    long    number; /* number of support routine */
    RECSUPFUN report; /* print report */
    RECSUPFUN init; /* init support */
    RECSUPFUN init_record; /* init record */
    RECSUPFUN process; /* process record */
    RECSUPFUN special; /* special processing */
    RECSUPFUN get_value; /* OBSOLETE: Just leave NULL */
    RECSUPFUN cvt_dbaddr; /* cvt dbAddr */
    RECSUPFUN get_array_info;
    RECSUPFUN put_array_info;
    RECSUPFUN get_units;
    RECSUPFUN get_precision;
    RECSUPFUN get_enum_str; /* get string from enum */
    RECSUPFUN get_enum_strs; /* get all enum strings */
    RECSUPFUN put_enum_str; /* put enum from string */
    RECSUPFUN get_graphic_double;
    RECSUPFUN get_control_double;
    RECSUPFUN get_alarm_double;
};
```

Each record support module must define its RSET. The external name must be of the form:

```
<record_type>RSET
```

Any routines not needed for the particular record type should be initialized to the value `NULL`. Look at the example below for details.

Device support routines are located via a Device Support Entry Table (DSET), which has the following structure:

```

struct dset { /* device support entry table */
    long      number; /* number of support routines */
    DEVSUPFUN report; /* print report */
    DEVSUPFUN init;   /* init support */
    DEVSUPFUN init_record; /* init record instance*/
    DEVSUPFUN get_ioint_info; /* get io interrupt info*/
    /* other functions are record dependent*/
};

```

Each device support module must define its associated DSET. The external name must be the same as the name which appears in devSup.ascii.

Any record support module which has associated device support must also include definitions for accessing its associated device support modules. The field "dset", which is located in dbCommon, contains the address of the DSET. It is given a value by iocInit.

11.4 Example Record Support Module

This section contains the skeleton of a record support package. The record type is xxx and the record has the following fields in addition to the dbCommon fields: VAL, PREC, EGU, HOPR, LOPR, HIHI, LOLO, HIGH, LOW, HHSV, LLSV, HSV, LSV, HYST, ADEL, MDEL, LALM, ALST, MLST. These fields will have the same meaning as they have for the ai record. Consult the Record Reference manual for a description.

11.4.1 Declarations

```

/* Create RSET - Record Support Entry Table*/
#define report NULL
#define initialize NULL
static long init_record();
static long process();
#define special NULL
#define get_value NULL
#define cvt_dbaddr NULL
#define get_array_info NULL
#define put_array_info NULL
static long get_units();
static long get_precision();
#define get_enum_str NULL
#define get_enum_strs NULL
#define put_enum_str NULL
static long get_graphic_double();
static long get_control_double();
static long get_alarm_double();

rset xxxRSET={
    RSETNUMBER,
    report,
    initialize,
    init_record,

```

```
process,
special,
get_value,
cvt_dbaddr,
get_array_info,
put_array_info,
get_units,
get_precision,
get_enum_str,
get_enum_strs,
put_enum_str,
get_graphic_double,
get_control_double,
get_alarm_double
};
epicsExportAddress(rset,xxxRSET);

/* declarations for associated DSET */
typedef struct xxxdset { /* analog input dset */
    long    number;
    DEVSUPFUN    dev_report;
    DEVSUPFUN    init;
    DEVSUPFUN    init_record; /* returns: (1,0)=> (failure, success)*/
    DEVSUPFUN    get_ioint_info;
    DEVSUPFUN    read_xxx;
}xxxdset;

/* forward declaration for internal routines*/
static void checkAlarams(xxxRecord *pxxx);
static void monitor(xxxRecord *pxxx);
```

The above declarations define the Record Support Entry Table (RSET), a template for the associated Device Support Entry Table (DSET), and forward declarations to private routines.

The RSET must be declared with an external name of xxxRSET. It defines the record support routines supplied for this record type. Note that forward declarations are given for all routines supported and a NULL declaration for any routine not supported.

The template for the DSET is declared for use by this module.

11.4.2 init_record

```
static long init_record(void *precord, int pass)
{
    xxxRecord*pxxx = (xxxRecord *)precord;
    xxxdset    *pdset;
    long    status;

    if(pass==0) return(0);

    if((pdset = (xxxdset *) (pxxx->dset)) == NULL) {
        recGblRecordError(S_dev_noDSET,pxxx,"xxx: init_record");
        return(S_dev_noDSET);
    }
}
```

```

/* must have read_xxx function defined */
if( (pdset->number < 5) || (pdset->read_xxx == NULL) ) {
    recGblRecordError(S_dev_missingSup,pxxx,
        "xxx: init_record");
    return(S_dev_missingSup);
}
if( pdset->init_record ) {
    if((status=(*pdset->init_record)(pxxx))) return(status);
}
return(0);
}

```

This routine, which is called by `iocInit` twice for each record of type `xxx`, checks to see if it has a proper set of device support routines and, if present, calls the `init_record` entry of the DSET.

During the first call to `init_record` (`pass=0`) only initializations relating to this record can be performed. During the second call (`pass=1`) initializations that may refer to other records can be performed. Note also that during the second pass, other records may refer to fields within this record. A good example of where these rules are important is a waveform record. The `VAL` field of a waveform record actually refers to an array. The waveform record support module must allocate storage for the array. If another record has a database link referring to the waveform `VAL` field then the storage must be allocated before the link is resolved. This is accomplished by having the waveform record support allocate the array during the first pass (`pass=0`) and having the link reference resolved during the second pass (`pass=1`).

11.4.3 process

```

static long process(void *precord)
{
    xxxRecord*pxxx = (xxxRecord *)precord;
    xxxdset *pdset = (xxxdset *)pxxx->dset;
    long status;
    unsigned char pact=pxxx->pact;

    if( (pdset==NULL) || (pdset->read_xxx==NULL) ) {
        /* leave pact true so that dbProcess doesnt call again*/
        pxxx->pact=TRUE;
        recGblRecordError(S_dev_missingSup,pxxx,"read_xxx");
        return(S_dev_missingSup);
    }

    /* pact must not be set true until read_xxx completes*/
    status=(*pdset->read_xxx)(pxxx); /* read the new value */
    /* return if beginning of asynch processing*/
    if(!pact && pxxx->pact) return(0);
    pxxx->pact = TRUE;
    recGblGetTimeStamp(pxxx);

    /* check for alarms */
    alarm(pxxx);
    /* check event list */
    monitor(pxxx);
    /* process the forward scan link record */
    recGblFwdLink(pxxx);
}

```

```
    pxxx->pact=FALSE;
    return(status);
}
```

The record processing routines are the heart of the IOC software. The record specific process routine is called by `dbProcess` whenever it decides that a record should be processed. Process decides what record processing really means. The above is a good example of what should be done. In addition to being called by `dbProcess` the process routine may also be called by asynchronous record completion routines.

The above model supports both synchronous and asynchronous device support routines. For example, if `read_xxx` is an asynchronous routine, the following sequence of events will occur:

- `process` is called with `pact FALSE`
- `read_xxx` is called. Since `pact` is `FALSE` it starts I/O, arranges callback, and sets `pact TRUE`
- `read_xxx` returns
- because `pact` went from `FALSE` to `TRUE` `process` just returns
- Any new call to `dbProcess` is ignored because it finds `pact TRUE`
- Sometime later the callback occurs and `process` is called again.
- `read_xxx` is called. Since `pact` is `TRUE` it knows that it is a completion request.
- `read_xxx` returns
- `process` completes record processing
- `pact` is set `FALSE`
- `process` returns

At this point the record has been completely processed. The next time `process` is called everything starts all over from the beginning.

11.4.4 Miscellaneous Utility Routines

```
static long get_units(DBADDR *paddr, char *units)
{
    xxxRecord *pxxx=(xxxRecord *)paddr->precord;

    strncpy(units,pxxx->egu,sizeof(pxxx->egu));
    return(0);
}
```

```
static long get_graphic_double(DBADDR *paddr,
    struct dbr_grDouble *pgd)
{
    xxxRecord *pxxx=(xxxRecord *)paddr->precord;
    int fieldIndex = dbGetFieldIndex(paddr);

    if(fieldIndex == xxxRecordVAL) {
        pgd->upper_disp_limit = pxxx->hopr;
        pgd->lower_disp_limit = pxxx->lopr;
    } else recGblGetGraphicDouble(paddr,pgd);
    return(0);
}
/* similar routines would be provided for */
/* get_control_double and get_alarm_double*/
```

These are a few examples of various routines supplied by a typical record support package. The functions that must be performed by the remaining routines are described in the next section.

11.4.5 Alarm Processing

```
static void checkAlarms(xxxRecord *pxxx)
{
    double          val;
    float           hyst,lalm,hihi,high,low,lolo;
    unsigned short hhsv,llsv,hsv,lsv;

    if(pxxx->udf == TRUE ){
        recGblSetSevr(pxxx,UDF_ALARM,VALID_ALARM);
        return;
    }

    hihi=pxxx->hihi; lolo=pxxx->lolo;
    high=pxxx->high; low=pxxx->low;
    hhsv=pxxx->hhsv; llsv=pxxx->llsv;
    hsv=pxxx->hsv; lsv=pxxx->lsv;
    val=pxxx->val; hyst=pxxx->hyst; lalm=pxxx->lalm;

    /* alarm condition hihi */
    if (hhsv && (val >= hihi
    || ((lalm==hihi) && (val >= hihi-hyst)))) {
        if(recGblSetSevr(pxxx,HIHI_ALARM,pxxx->hhsv)
        pxxx->lalm = hihi;
        return;
    }
    /* alarm condition lolo */
    if (llsv && (val <= lolo
    || ((lalm==lolo) && (val <= lolo+hyst)))) {
        if(recGblSetSevr(pxxx,LOLO_ALARM,pxxx->llsv))
        pxxx->lalm = lolo;
        return;
    }
    /* alarm condition high */
    if (hsv && (val >= high
    || ((lalm==high) && (val >= high-hyst)))) {
        if(recGblSetSevr(pxxx,HIGH_ALARM,pxxx->hsv))
        pxxx->lalm = high;
        return;
    }
    /* alarm condition low */
    if (lsv && (val <= low
    || (lalm==low) && (val <= low+hyst)))) {
        if(recGblSetSevr(pxxx,LOW_ALARM,pxxx->lsv))
        pxxx->lalm = low;
        return;
    }
    /*we get here only if val is out of alarm by at least hyst*/
    pxxx->lalm=val;
}
```

```
    return;  
}
```

This is a typical set of code for checking alarms conditions for an analog type record. The actual set of code can be very record specific. Note also that other parts of the system can raise alarms. The algorithm is to always maximize alarm severity, i.e. the highest severity outstanding alarm will be reported.

The above algorithm also honors a hysteresis factor for the alarm. This is to prevent alarm storms from occurring in the event that the current value is very near an alarm limit and noise makes it continually cross the limit. It honors the hysteresis only when the value is going to a lower alarm severity.

Note the test:

```
if(pxxx->udf == TRUE ){  
    recGblSetSevr(pxxx,UDF_ALARM,VALID_ALARM);  
    return;  
}
```

Database common defines the field UDF, which means that field VAL is undefined. The STAT and SEVR fields are initialized as though `recGblSetSevr(pxxx,UDF_ALARM,VALID_ALARM)` was called. Thus if the record is never processed the record will be in an INVALID UNDEFINED alarm state. Field UDF is initialized to the value 1, i.e. TRUE. Thus the above code will keep the record in the INVALID UNDEFINED alarm state as long as UDF is not given the value 0.

The UDF field means Undefined, i.e. the VAL field has never been given a value. When records are loaded into an ioc this is the initial state of records. Whenever code gives a value to the VAL field it is also supposed to set UDF false. Unless a particular record type has unusual semantics no code should set UDF true. UDF normally means that the field was never given a value.

For input records device support is responsible for obtaining an input value. If no input value can be obtained neither record support nor device support sets UDF false. If device support reads a raw value it returns a value telling record support to perform a conversion. After the record support sets VAL equal to the converted value, it sets UDF false. If device support obtains a converted value that it writes to VAL, it sets UDF false.

For output records either something outside record/device support writes to the VAL field or else VAL is given a value because record support obtains a value via the OMSL field. In either case the code that writes to the VAL field sets UDF false.

Whenever database access writes to the VAL field it sets UDF false.

Routine `recGblSetSevr` is called to raise alarms. It can be called by `iocCore`, record support, or device support. The code that detects an alarm is responsible for raising the alarm.

11.4.6 Raising Monitors

```
static void monitor(xxxRecord *pxxx)  
{  
    unsigned short    monitor_mask;  
    float             delta;  
  
    monitor_mask = recGblResetAlarms(pxxx);  
    /* check for value change */  
    delta = pxxx->mlst - pxxx->val;  
    if(delta<0.0) delta = -delta;  
    if (delta > pxxx->mdel) {  
        /* post events for value change */  
        monitor_mask |= DBE_VALUE;  
    }  
}
```

```

    /* update last value monitored */
    pxxx->mlst = pxxx->val;
}
/* check for archive change */
delta = pxxx->alst - pxxx->val;
if(delta<0.0) delta = 0.0;
if (delta > pxxx->adel) {
    /* post events on value field for archive change */
    monitor_mask |= DBE_LOG;
    /* update last archive value monitored */
    pxxx->alst = pxxx->val;
}
/* send out monitors connected to the value field */
if (monitor_mask){
    db_post_events(pxxx,&pxxx->val,monitor_mask);
}
return;
}

```

All record types should call `recGblResetAlarms` as shown. Note that `nsta` and `nsev` will have the value 0 after this routine completes. This is necessary to ensure that alarm checking starts fresh after processing completes. The code also takes care of raising alarm monitors when a record changes from an alarm state to the no alarm state. It is essential that record support routines follow the above model or else alarm processing will not follow the rules.

Analog type records should also provide monitor and archive hysteresis fields as shown by this example.

`db_post_events` results in channel access issuing monitors for clients attached to the record and field. The call is

```

int db_post_events(void *precord, void *pfield,
    unsigned int monitor_mask)

```

where:

```

precord - The address of the record
pfield - The address of the field
monitor_mask - A bit mask that can be any combinations of the following:
    DBE_ALARM - A change of alarm state has occurred. This is set by recGblResetAlarms.
    DBE_LOG - Archive change of state.
    DBE_VAL - Value change of state

```

IMPORTANT: The record support module is responsible for calling `db_post_event` for any fields that change as a result of record processing. Also it should **NOT** call `db_post_event` for fields that do not change.

11.5 Record Support Routines

This section describes the routines defined in the RSET. Any routine that does not apply to a specific record type must be declared `NULL`.

11.5.1 Generate Report of Each Field in Record

```

report(void *precord); /* addr of record*/

```

This routine is not used by most record types. Any action is record type specific.

11.5.2 Initialize Record Processing

```
initialize(void);
```

This routine is called once at IOC initialization time. Any action is record type specific. Most record types do not need this routine.

11.5.3 Initialize Specific Record

```
init_record(  
    void *precord, /* addr of record*/  
    int    pass);
```

iocInit calls this routine twice (pass=0 and pass=1) for each database record of the type handled by this routine. It must perform the following functions:

- Check and/or issue initialization calls for the associated device support routines.
- Perform any record type specific initialization.
- During the first pass it can only perform initializations that affect the record referenced by precord.
- During the second pass it can perform initializations that affect other records.

11.5.4 Process Record

```
process(void *precord); /* addr of record*/
```

This routine must follow the guidelines specified previously.

11.5.5 Special Processing

```
special(  
    struct dbAddr *paddr,  
    int    after); /* (FALSE, TRUE) => (Before, After) Processing*/
```

This routine implements the record type specific special processing for the field referred to by dbAddr. Note that it is called twice. Once before any changes are made to the associated field and once after. File special.h defines special types. This routine is only called for user special fields, i.e. fields with SPC_XXX >= 100. A field is declared special in the ASCII record definition file. New values should not be added to special.h, instead use SPC_MOD.

The database access routine, dbGetFieldIndex can be used to determine which field is being modified.

11.5.6 Get Value

This routine is no longer used. It should be left as a NULL procedure in the record support entry table.

11.5.7 Convert dbAddr Definitions

```
cvt_dbaddr(struct dbAddr *paddr);
```

This routine is called by `dbNameToAddr` if the field has special set equal to `SPC_DBADDR`. A typical use is when a field refers to an array. This routine can change any combination of the `dbAddr` fields: `no_elements`, `field_type`, `field_size`, `special`, `pfield`, and `dbr_type`. For example if the `VAL` field of a waveform record is passed to `dbNameToAddr`, `cvt_dbaddr` would change `dbAddr` so that it refers to the actual array rather than `VAL`.

The database access routine, `dbGetFieldIndex` can be used to determine which field is being modified.

NOTES:

- Channel access calls `db_name_to_addr`, which is part of old database access. `Db_name_to_addr` calls `dbNameToAddr`. This is done when a client connects to the record.
- `no_elements` must be set to the maximum number of elements that will ever be stored in the array.

11.5.8 Get Array Information

```
get_array_info(
    struct dbAddr  *paddr,
    long   *no_elements,
    long   *offset);
```

This routine returns the current number of elements and the offset of the first value of the specified array. The offset field is meaningful if the array is actually a circular buffer.

The database access routine, `dbGetFieldIndex` can be used to determine which field is being modified. It is permissible for `get_array_info` to change `pfield`. This feature can be used to implement double buffering.

When an array field is being written `get_array_info` is called before the field values are changed.

11.5.9 Put Array Information

```
put_array_info(
    struct dbAddr  *paddr,
    long   nNew);
```

This routine is called after new values have been placed in the specified array.

The database access routine, `dbGetFieldIndex` can be used to determine which field is being modified.

11.5.10 Get Units

```
get_units(
    struct dbAddr  *paddr,
    char  *punits);
```

This routine sets units equal to the engineering units for the field.

The database access routine, `dbGetFieldIndex` can be used to determine which field is being modified.

11.5.11 Get Precision

```
get_precision(
    struct dbAddr  *paddr,
    long   *precision);
```

This routine gets the precision, i.e. number of decimal places, which should be used to convert the field value to an ASCII string. `recGblGetPrec` should be called for fields not directly related to the value field.

The database access routine, `dbGetFieldIndex` can be used to determine which field is being modified.

11.5.12 Get Enumerated String

```
get_enum_str(  
    struct dbAddr  *paddr,  
    char  *p);
```

This routine sets `*p` equal to the ASCII string for the field value. The field must have type `DBF_ENUM`.

Look at the code for the `bi` or `mbbi` records for examples.

The database access routine, `dbGetFieldIndex` can be used to determine which field is being modified.

11.5.13 Get Strings for Enumerated Field

```
get_enum_strs(  
    struct dbAddr  *paddr,  
    struct dbr_enumStrs  *p);
```

This routine gives values to all fields of structure `dbr_enumStrs`.

Look at the code for the `bi` or `mbbi` records for examples.

The database access routine, `dbGetFieldIndex` can be used to determine which field is being modified.

11.5.14 Put Enumerated String

```
put_enum_str(  
    struct dbAddr  *paddr,  
    char  *p);
```

Given an ASCII string, this routine updates the database field. It compares the string with the string values associated with each enumerated value and if it finds a match sets the database field equal to the index of the string which matched.

Look at the code for the `bi` or `mbbi` records for examples.

The database access routine, `dbGetFieldIndex` can be used to determine which field is being modified.

11.5.15 Get Graphic Double Information

```
get_graphic_double(  
    struct dbAddr  *paddr,  
    struct dbr_grDouble  *p); /* addr of return info*/
```

This routine fills in the graphics related fields of structure `dbr_grDouble`. `recGblGetGraphicDouble` should be called for fields not directly related to the value field.

The database access routine, `dbGetFieldIndex` can be used to determine which field is being modified.

11.5.16 Get Control Double Information

```
get_control_double(
    struct dbAddr *paddr,
    struct dbr_ctrlDouble *p); /* addr of return info*/
```

This routine gives values to all fields of structure `dbr_ctrlDouble`. `recGblGetControlDouble` should be called for fields not directly related to the value field.

The database access routine, `dbGetFieldIndex` can be used to determine which field is being modified.

11.5.17 Get Alarm Double Information

```
get_alarm_double(
    struct dbAddr *paddr,
    struct dbr_alDouble *p); /* addr of return info*/
```

This routine gives values to all fields of structure `dbr_alDouble`.

The database access routine, `dbGetFieldIndex` can be used to determine which field is being modified.

11.6 Global Record Support Routines

A number of global record support routines are available. These routines are intended for use by the record specific processing routines but can be called by any routine that wishes to use their services.

The name of each of these routines begins with "recGbl".

11.6.1 Alarm Status and Severity

Alarms may be raised in many different places during the course of record processing. The algorithm is to maximize the alarm severity, i.e. the highest severity outstanding alarm is raised. If more than one alarm of the same severity is found then the first one is reported. This means that whenever a code fragment wants to raise an alarm, it does so only if the alarm severity it will declare is greater than that already existing. Four fields (in database common) are used to implement alarms: `sevr`, `stat`, `nsev`, and `nsta`. The first two are the status and severity after the record is completely processed. The last two fields (`nsta` and `nsev`) are the status and severity values to set during record processing. Two routines are used for handling alarms. Whenever a routine wants to raise an alarm it calls `recGblSetSevr`. This routine will only change `nsta` and `nsev` if it will result in the alarm severity being increased. At the end of processing, the record support module must call `recGblResetAlarms`. This routine sets `stat=nsta`, `sevr=nsev`, `nsta=0`, and `nsev=0`. If `stat` or `sevr` has changed value since the last call it calls `db_post_event` for `stat` and `sevr` and returns a value of `DBE_ALARM`. If no change occurred it returns 0. Thus after calling `recGblResetAlarms` everything is ready for raising alarms the next time the record is processed. The example record support module presented above shows how these macros are used.

```
recGblSetSevr(
    void *precord,
    short nsta,
    short nsevr);
```

Returns: (TRUE, FALSE) if (did, did not) change `nsta` and `nsev`.

```
unsigned short recGblResetAlarms(void *precord);
```

Returns: Initial value for `monitor_mask`

11.6.2 Alarm Acknowledgment

Database common contains two additional alarm related fields: `acks` (Highest severity unacknowledged alarm) and `ackt` (does transient alarm need to be acknowledged). These field are handled by `iocCore` and `recGblResetAlarms` and are not the responsibility of record support. These fields are intended for use by the alarm handler.

11.6.3 Generate Error: Process Variable Name, Caller, Message

SUGGESTION: use `errlogPrintf` instead of this for new code.

```
recGblDbaddrError(  
    long    status,  
    struct dbAddr  *paddr,  
    char    *pcaller_name); /* calling routine name */
```

This routine interfaces with the system wide error handling system to display the following information: Status information, process variable name, calling routine.

11.6.4 Generate Error: Status String, Record Name, Caller

SUGGESTION: use `errlogPrintf` instead of this for new code.

```
recGblRecordError(  
    long    status,  
    void    *precord, /* addr of record */  
    char    *pcaller_name); /* calling routine name */
```

This routine interfaces with the system wide error handling system to display the following information: Status information, record name, calling routine.

11.6.5 Generate Error: Record Name, Caller, Record Support Message

SUGGESTION: use `errlogPrintf` instead of this for new code.

```
recGblRecsupError(  
    long    status,  
    struct dbAddr  *paddr,  
    char    *pcaller_name, /* calling routine name */  
    char    *psupport_name); /* support routine name*/
```

This routine interfaces with the system wide error handling system to display the following information: Status information, record name, calling routine, record support entry name.

11.6.6 Get Graphics Double

```
recGblGetGraphicDouble(  
    struct dbAddr  *paddr,  
    struct dbr_grDouble  *pgd);
```

This routine can be used by the `get_graphic_double` record support routine to obtain graphics values for fields that it doesn't know how to set.

11.6.7 Get Control Double

```
recGblGetControlDouble(
    struct dbAddr *paddr,
    struct dbr_ctrlDouble *pcd);
```

This routine can be used by the `get_control_double` record support routine to obtain control values for fields that it doesn't know how to set.

11.6.8 Get Alarm Double

```
recGblGetAlarmDouble(
    struct dbAddr *paddr,
    struct dbr_alDouble *pcd);
```

This routine can be used by the `get_alarm_double` record support routine to obtain control values for fields that it doesn't know how to set.

11.6.9 Get Precision

```
recGblGetPrec(
    struct dbAddr *paddr,
    long *pprecision);
```

This routine can be used by the `get_precision` record support routine to obtain the precision for fields that it doesn't know how to set the precision.

11.6.10 Get Time Stamp

```
recGblGetTimeStamp(void *precord)
```

This routine gets the current time stamp and puts it in the record. It does the following:

- If TSEL is not a constant link and TSEL refers to the TIME field of a record, the time is obtained from the record reference by TSEL and this is put in field TIME. The routine then returns.
- If TSEL is not a constant link `dbGetLink` is called and the value is put in field TSE.
- If TSE is equal to `epicsTimeEventDeviceTime` (-2) then nothing is done, i.e. the routine just returns.
- `epicsTimeGetEvent` is called.

11.6.11 Forward link

```
recGblFwdLink(
    void *precord);
```

This routine can be used by process to request processing of forward links.

11.6.12 Initialize Constant Link

```
int recGblInitConstantLink(
    struct link *plink,
    short dbfType,
    void *pdest);
```

Initialize a constant link. This routine is usually called by `init_record` (or by associated device support) to initialize the field associated with a constant link. It returns (FALSE, TRUE) if it (did not, did) modify the destination.

Chapter 12: Device Support

12.1 Overview

In addition to a record support module, each record type can have an arbitrary number of device support modules. The purpose of device support is to hide hardware specific details from record processing routines. Thus support can be developed for a new device without changing the record support routines.

A device support routine has knowledge of the record definition. It also knows how to talk to the hardware directly or how to call a device driver which interfaces to the hardware. Thus device support routines are the interface between hardware specific fields in a database record and device drivers or the hardware itself.

Release 3.14.8 introduced the concept of extended device support, which provides an optional interface that a device support can implement to obtain notification when a record's address is changed at runtime. This permits records to be reconnected to a different kind of I/O device, or just to a different signal on the same device. Extended device support is described in more detail in Section 12.5 on page 186 below.

Database common contains two device related fields:

- **dtyp**: Device Type.
- **dset**: Address of Device Support Entry Table.

The field `dtyp` contains the index of the menu choice as defined by the device ASCII definitions. `iocInit` uses this field and the device support structures defined in `devSup.h` to initialize the field `dset`. Thus record support can locate its associated device support via the `dset` field.

Device support modules can be divided into two basic classes: synchronous and asynchronous. Synchronous device support is used for hardware that can be accessed without delays for I/O. Many register based devices are synchronous devices. Other devices, for example all GPIB devices, can only be accessed via I/O requests that may take large amounts of time to complete. Such devices must have associated asynchronous device support. Asynchronous device support makes it more difficult to create databases that have linked records.

If a device can be accessed with a delay of less than a few microseconds then synchronous device support is appropriate. If a device causes delays of greater than 100 microseconds then asynchronous device support is appropriate. If the delay is between these values your guess about what to do is as good as mine. Perhaps you should ask the hardware designer why such a device was created.

If a device takes a long time to accept requests there is another option than asynchronous device support. A driver can be created that periodically polls all its attached input devices. The device support just returns the latest polled value. For outputs, device support just notifies the driver that a new value must be written. the driver, during one of its polling phases, writes the new value. The EPICS Allen Bradley device/driver support is a good example.

12.2 Example Synchronous Device Support Module

```
/* Create the dset for devAiSoft */  
long init_record();
```

```
long read_ai();
struct {
    long    number;
    DEVSUPFUN    report;
    DEVSUPFUN    init;
    DEVSUPFUN    init_record;
    DEVSUPFUN    get_ioint_info;
    DEVSUPFUN    read_ai;
    DEVSUPFUN    special_linconv;
}devAiSoft={
    6,
    NULL,
    NULL,
    init_record,
    NULL,
    read_ai,
    NULL
};
epicsExportAddress(dset,devAiSoft);

static long init_record(void *precord)
{
    aiRecord    *pai = (aiRecord *)precord;
    long status;

    /* ai.inp must be a CONSTANT, PV_LINK, DB_LINK or CA_LINK*/
    switch (pai->inp.type) {
        case (CONSTANT) :
            if(recGblInitConstantLink(&pai->inp,DBF_DOUBLE,&pai->val))
                pai->udf = FALSE;
            break;
        case (PV_LINK) :
        case (DB_LINK) :
        case (CA_LINK) :
            break;
        default :
            recGblRecordError(S_db_badField, (void *)pai,
                "devAiSoft (init_record) Illegal INP field");
            return(S_db_badField);
    }
    /* Make sure record processing routine does not perform any conversion*/
    pai->linr=0;
    return(0);
}

static long read_ai(void *precord)
{
    aiRecord*pai =(aiRecord *)precord;
    long status;

    status=dbGetGetLink(&(pai->inp.value.db_link),
        (void *)pai,DBR_DOUBLE,&(pai->val),0,1);
    if (pai->inp.type!=CONSTANT && RTN_SUCCESS(status)) pai->udf = FALSE;
}
```

```

    return(2); /*don't convert*/
}

```

The example is `devAiSoft`, which supports soft analog inputs. The `INP` field can be a constant or a database link or a channel access link. Only two routines are provided (the rest are declared `NULL`). The `init_record` routine first checks that the link type is valid. If the link is a constant it initializes `VAL`. If the link is a Process Variable link it calls `dbCaGetLink` to turn it into a Channel Access link. The `read_ai` routine obtains an input value if the link is a database or Channel Access link, otherwise it doesn't have to do anything.

12.3 Example Asynchronous Device Support Module

This example shows how to write an asynchronous device support routine. It does the following sequence of operations:

1. When first called `pact` is `FALSE`. It arranges for a callback (`myCallback`) routine to be called after a number of seconds specified by the `DISV` field.
2. It prints a message stating that processing has started, sets `pact` `TRUE`, and returns. The record processing routine returns without completing processing.
3. When the specified time elapses `myCallback` is called. It calls `dbScanLock` to lock the record, calls `process`, and calls `dbScanUnlock` to unlock the record. It calls the `process` entry of the record support module, which it locates via the `rset` field in `dbCommon`, directly rather than `dbProcess`. `dbProcess` would not call `process` because `pact` is `TRUE`.
4. When `process` executes, it again calls `read_ai`. This time `pact` is `TRUE`.
5. `read_ai` prints a message stating that record processing is complete and returns a status of 2. Normally a value of 0 would be returned. The value 2 tells the record support routine not to attempt any conversions. This is a convention (a bad convention!) used by the analog input record.
6. When `read_ai` returns the record processing routine completes record processing.

At this point the record has been completely processed. The next time `process` is called everything starts all over.

Note that this is somewhat of an artificial example since real code of this form would more likely use the `callbackcallbackRequestProcessCallbackDelayed` function to perform the required processing.

```

static void myCallback(CALLBACK *pcallback)
{
    struct dbCommon *precord;
    struct rset      *prset;

    callbackGetUser(precord,pcallback);
    prset=(struct rset *) (precord->rset);
    dbScanLock(precord);
    (*prset->process)(precord);
    dbScanUnlock(precord);
}

static long init_record(struct aiRecord *pai)
{
    CALLBACK *pcallback;
    switch (pai->inp.type) {
    case (CONSTANT) :
        pcbcallback = (CALLBACK *) (calloc(1,sizeof(CALLBACK)));
        callbackSetCallback(myCallback,pcbcallback);
        callbackSetUser(pai,pcbcallback);

```

```
        pai->dpvt = (void *)pcallback;
        break;
    default :
        recGblRecordError(S_db_badField,(void *)pai,
            "devAiTestAsyn (init_record) Illegal INP field");
        return(S_db_badField);
    }
    return(0);
}

static long read_ai(struct aiRecord *pai)
{
    CALLBACK *pcallback = (CALLBACK *)pai->dpvt;
    if(pai->pact) {
        pai->val += 0.1; /* Change VAL just to show we've done something. */
        pai->udf = FALSE; /* We modify VAL so we are responsible for UDF too. */
        printf("Completed asynchronous processing: %s\n",pai->name);
        return(2); /* don't convert*/
    }
    printf("Starting asynchronous processing: %s\n",pai->name);
    pai->pact=TRUE;
    callbackRequestDelayed(pcallback,pai->disv);
    return(0);
}

/* Create the dset for devAiTestAsyn */
struct {
    long          number;
    DEVSUPFUN    report;
    DEVSUPFUN    init;
    DEVSUPFUN    init_record;
    DEVSUPFUN    get_ioint_info;
    DEVSUPFUN    read_ai;
    DEVSUPFUN    special_linconv;
}devAiTestAsyn={
    6,
    NULL,
    NULL,
    init_record,
    NULL,
    read_ai,
    NULL
};
epicsExportAddress(dset,devAiTestAsyn);
```

12.4 Device Support Routines

This section describes the routines defined in the DSET. Any routine that does not apply to a specific record type must be declared NULL.

12.4.1 Generate Device Report

```
long report(  
    int    interest);
```

This routine is responsible for reporting all I/O cards it has found. If `interest` is (0,1) then generate a (short, long) report. If a device support module is using a driver, it normally does not have to implement this routine because the driver generates the report.

12.4.2 Initialize Device Processing

```
long init(  
    int    after);
```

This routine is called twice at IOC initialization time. Any action is device specific. This routine is called twice: once before any database records are initialized and once after all records are initialized but before the scan tasks are started. `after` has the value (0,1) (before, after) record initialization.

12.4.3 Initialize Specific Record

```
long init_record(  
    void *precord); /* addr of record*/
```

The record support `init_record` routine calls this routine.

12.4.4 Get I/O Interrupt Information

```
long get_ioint_info(  
    int    cmd,  
    struct dbCommon *precord,  
    IOSCANPVT *ppvt);
```

This is called by the I/O interrupt scan task. If `cmd` is (0,1) then this routine is being called when the associated record is being (placed in, taken out of) an I/O scan list. See the chapter on scanning for details.

It should be noted that a previous type of I/O event scanning is still supported. It is not described in this document because, hopefully, it will go away in the near future. When it calls this routine the arguments have completely different meanings.

12.4.5 Other Device Support Routines

All other device support routines are record type specific.

12.5 Extended Device Support

This section describes the additional behaviour and routines required for a device support layer to support online changes to a record's hardware address.

12.5.1 Rationale

In releases prior to R3.14.8 it is possible to change the value of the INP or OUT field of a record but (unless a soft device support is in use) this generally has no effect on the behaviour of the device support at all. Some device supports have been written that check this hardware address field for changes every time they process, but they are in the minority and in any case they do not provide any means to switch between different device support layers at runtime since no software is present that can lookup a new value for the DSET field after `iocInit`.

The extended device interface has been carefully designed to retain maximal backwards compatibility with existing device and record support layers, and as a result it cannot just introduce new routines into the DSET:

- Different record types have different numbers of DSET routines
- Every device support layer defines its own DSET structure layout
- Some device support layers add their own routines to the DSET (GPIB, BitBus)

Since both basic and extended device support layers have to co-exist within the same IOC, some rules are enforced concerning whether the device address of a particular record is allowed to be changed:

1. Records that were connected at `iocInit` to a device support layer that does not implement the extended interface are never allowed to have address fields changed at runtime.
2. Extended device support layers are not required to implement both the `add_record` and `del_record` routines, thus some devices may only allow one-way changes.
3. The old device support layer is informed and allowed to refuse an address change before the field change is made (it does not get to see the new address).
4. The new device support layer is informed after the field change has been made, and it may refuse to accept the record. In this case the record will be set as permanently busy (`PACT=true`) until an address is accepted.
5. Record support layers can also get notified about this process by making their address field special, in which case the record type's special routine can refuse to accept the new address before it is presented to the device support layer. Special cannot prevent the old device support from being disconnected however.

If an address change is refused, the change to the INP or OUT field will cause an error or exception to be passed to the software performing the change. If this was a Channel Access client the result is to generate an exception callback.

To switch to a different device support layer, it is necessary to change the DTYP field before the INP or OUT field. The change to the DTYP field has no effect until the latter field change takes place.

If a record is set to I/O Interrupt scan but the new layer does not support this, the scan will be changed to Passive.

12.5.2 Initialization/Registration

Device support that implements the extended behaviour must provide an `init` routine in the Device Support Entry Table (see Section 12.4.2 on page 185). In the first call to this routine (pass 0) it registers the address of its Device Support eXtension Table (DSXT) in a call to `devExtend`.

The only exception to this registration requirement is when the device support uses a link type of `CONSTANT`. In this circumstance the system will automatically register an empty DSXT for that particular support layer (both the `add_record` and `del_record` routines pointed to by this DSXT do nothing and return zero). This exception allows existing soft channel device support layers to continue to work without requiring any modification, since the `iocCore` software already takes care of changes to `PV_LINK` addresses.

The following is an example of a DSXT and the initialization routine that registers it:

```
static struct dsxt myDsxt = {
    add_record, del_record
};

static long init(int pass) {
    if (pass==0) devExtend(&myDsxt);
    return 0;
}
```

A call to `devExtend` can only be made during the first pass of the device support initialization process, and registers the DSXT for that device support layer; if called at any other time it will log an error message and immediately return.

12.5.3 Device Support eXtension Table

The full definition of `struct dsxt` is found in `devSup.h` and currently looks like this:

```
typedef struct dsxt {
    long (*add_record)(struct dbCommon *precord);
    long (*del_record)(struct dbCommon *precord);
} dsxt;
```

There may be future additions to this table to support additional functionality; such extensions may only be made by changing the `devSup.h` header file and rebuilding EPICS Base and all support modules, thus neither record types nor device support are permitted to make any private use of this table.

The two function pointers are the means by which the extended device support is notified about the record instances it is being given or that are being moved away from its control. In both cases the only parameter is a pointer to the record concerned, which the code will have to cast to the appropriate pointer for the record type. The return value from the routines should be zero for success, or an EPICS error status code.

12.5.4 Add Record Routine

```
long add_record(
    struct dbCommon *precord);
```

This function is called to offer a new record to the device support. It is also called during `iocInit`, in between the pass 0 and pass 1 calls to the regular device support `init_record` routine (described in Section 12.4.3 on page 185 above). When converting an existing device support layer, this routine will usually be very similar to the old `init_record` routine, although in some cases it may be necessary to do a little more work depending on the particular record type involved. The extra code required in these cases can generally be copied straight from the record type implementation itself. This is necessary because the record type has no knowledge of the address change that is taking place, so the device support must perform any bitmask generation and/or readback value conversions itself. This document does not attempt to describe all the necessary processing for the various different standard record types, although the following (incomplete) list is presented as an aid to device support authors:

- `mbbi/mbbo` record types: Set SHFT, convert NOBT and SHFT into MASK
- `bi/bo` record types: Set SHFT, convert SHFT to MASK
- analog record types: Calculate ESLO and EOFF
- Output record types: Possibly read the current value from hardware and back-convert to VAL, or send the current record output value to the hardware. *This behaviour is not required or defined, and it's not obvious what should be done. There may be complications here with ao records using OROC and/or OIF=Incremental; solutions to this issue have yet to be considered by the community.*

If the `add_record` routine discovers any errors, say in the link address, it should return a non-zero error status value to reject the record. This will cause the record's PACT field to be set, preventing any further processing of this record until some other address change to it gets accepted.

12.5.5 Delete Record Routine

```
long del_record(  
    struct dbCommon *precord);
```

This function is called to notify the device support of a request to change the hardware address of a record, and allow the device support to free up any resources it may have dedicated to this particular record.

Before this routine is called, the record will have had its SCAN field changed to Passive if it had been set to I/O Interrupt. This ensures that the device support's `get_ioint_info` routine is never called after the the call to `del_record` has returned successfully, although it may also lead to the possibility of missed interrupts if the address change is rejected by the `del_record` routine.

If the device support is unable to disconnect from the hardware for some reason, this routine should return a non-zero error status value, which will prevent the hardware address from being changed. In this event the SCAN field will be restored if it was originally set to I/O Interrupt.

After a successfull call to `del_record`, the record's DPVT field is set to NULL and PACT is cleared, ready for use by the new device support.

12.5.6 Init Record Routine

The `init_record` routine from the DSET (Section 12.4.3 on page 185) is called by the record type, and must still be provided since the record type's per-record initialization is run some time *after* the initial call to the DSXT's `add_record` routine. Most record types perform some initialization of record fields at this point, and an extended device support layer may have to fix anything that the record overwrites. The following (incomplete) list is presented as an aid to device support authors:

- mbbi/mbbo record types: Calculate MASK from SHFT
- analog record types: Calculate ESLO and EOFF
- Output record types: Perform readback of the initial raw value from the hardware.

Chapter 13: Driver Support

13.1 Overview

It is not necessary to create a driver support module in order to interface EPICS to hardware. For simple hardware device support is sufficient. At the present time most hardware support has both. The reason for this is historical. Before EPICS there was GTACS. During the change from GTACS to EPICS, record support was changed drastically. In order to preserve all existing hardware support the GTACS drivers were used without change. The device support layer was created just to shield the existing drivers from the record support changes.

Since EPICS now has both device and driver support the question arises: When do I need driver support and when don't I? Lets give a few reasons why drivers should be created.

- The hardware is actually a subnet, e.g. GPIB. In this case a driver should be provided for accessing the subnet. There is no reason to make the driver aware of EPICS except possibly for issuing error messages.
- The hardware is complicated. In this case supplying driver support helps modularized the software. The Allen Bradley driver, which is also an example of supporting a subnet, is a good example.
- An existing driver, maintained by others, is available. I don't know of any examples.
- The driver should be general purpose, i.e. not tied to EPICS. The CAMAC driver is a good example. It is used by other systems, such as CODA. This is perhaps the most important reason for driver support.
- For common devices, e.g. GPIB, CAN, CAMAC, etc. a generic driver layer should be created. This generic layer should be independent of EPICS and independent of low level interfaces. It should also define an interface for low level drivers. This allows low level interfaces to be replaced without impacting IOC records, record support, or device support.

The only thing needed to interface a driver to EPICS is to provide a driver support module, which can be layered on top of an existing driver, and provide a database definition for the driver. The driver support module is described in the next section. The database definition is described in chapter "Database Definition".

13.2 Device Drivers

Device drivers are modules that interface directly with the hardware. They are provided to isolate device support routines from details of how to interface to the hardware. Device drivers have no knowledge of the internals of database records. Thus there is no necessary correspondence between record types and device drivers. For example the Allen Bradley driver provides support for many different types of signals including analog inputs, analog outputs, binary inputs, and binary outputs.

In general only device support routines know how to call device drivers. Since device support varies widely from device to device, the set of routines provided by a device driver is almost completely driver dependent. The only requirement is that routines `report` and `init` must be provided. Device support routines must, of course, know what routines are provided by a driver.

File `drvSup.h` describes the format of a driver support entry table. The driver support module must supply a driver entry table. An example definition is:

```
LOCAL long report();
LOCAL long init();
struct {
    long    number;
    DRVSUPFUN    report;
    DRVSUPFUN    init;
} drvAb={
    2,
    report,
    init
};
epicsExportAddress(drvet,drvGpib);
```

The above example is for the Allen Bradley driver. It has an associated ascii definition of:

```
driver(drvGpib)
```

Thus it is seen that the driver support module should supply two EPICS callable routines: `int` and `report`.

13.2.0.1 init

This routine, which has no arguments, is called by `iocInit`. The driver is expected to look for and initialize the hardware it supports. As an example the `init` routine for Allen Bradley is:

```
LOCAL long init()
{
    return(ab_driver_init());
}
```

13.2.0.2 report

The `report` routine is called by the `dbior`, an IOC test routine. It is responsible for producing a report describing the hardware it found at `init` time. It is passed one argument, `level`, which is a hint about how much information to display. An example, taken from Allen Bradley, is:

```
LOCAL long report(int level)
{
    return(ab_io_report(level));
}
```

Guidelines for `level` are as follows:

Level=0	Display a one line summary for each device
Level=1	Display more information
Level=2	Display a lot of information. It is even permissible to prompt for what is wanted.

13.2.0.3 Hardware Configuration

Hardware configuration includes the following:

- VME/VXI address space
- VME Interrupt Vectors and levels
- Device Specific Information

The information contained in hardware links supplies some but not all configuration information. In particular it does not define the VME/VXI addresses and interrupt vectors. This additional information is what is meant by hardware configuration in this chapter.

The problem of defining hardware configuration information is an unsolved problem for EPICS. At one time configuration information was defined in `module_types.h`. Many existing device/driver support modules still use this method. It should **NOT** be used for any new support for the following reasons:

- There is no way to manage this file for the entire EPICS community.
- It does not allow arbitrary configuration information.
- It is hard for users to determine what the configuration information is.

The fact that it is now easy to include ASCII definitions for only the device/driver support used in each IOC makes the configuration problem much more manageable than previously. Previously if you wanted to support a new VME module it was necessary to pick addresses that nothing in `module_types.h` was using. Now you only have to check modules you are actually using.

Since there are no EPICS defined rules for hardware configuration, the following minimal guidelines should be used:

- Never use `#define` to specify things like VME addresses. Instead use variables and assign default values. Allow the default values to be changed before `iocInit` is executed. The best way is to supply a global routine that can be invoked from the IOC startup file. Note that all arguments to such routines should be one of the following:

```
int
char *
double
```

- Call the routines described in chapter “Device Support Library” whenever possible.

Chapter 14: Static Database Access

14.1 Overview

An IOC database is created on a Unix system via a Database Configuration Tool and stored in a Unix file. EPICS provides two sets of database access routines: Static Database Access and Runtime Database Access. Static database access can be used on Unix or IOC database files. Runtime database requires an initialized IOC databases. Static database access is described in this chapter and runtime database access in the next chapter.

Static database access provides a simplified interface to a database, i.e. much of the complexity is hidden. `DBF_MENU` and `DBF_DEVICE` fields are accessed via a common type called `DCT_MENU`. A set of routines are provided to simplify access to link fields. All fields can be accessed as character strings. This interface is called static database access because it can be used to access an uninitialized as well as an initialized database.

Before accessing database records, the files describing menus, record types, and devices must be read via `dbReadDatabase` or `dbReadDatabaseFP`. These routines, which are also used to load record instances, can be called multiple times.

Database Configuration Tools (DCTs) should manipulate an EPICS database only via the static database access interface. An IOC database is created on a host system via a database configuration tool and stored in a host file with a file extension of `".db"`. Three routines (`dbReadDatabase`, `dbReadDatabaseFP` and `dbWriteRecord`) access the host database file. These routines read/write a database file to/from a memory resident EPICS database. All other access routines manipulate the memory resident database.

An include file `dbStaticLib.h` contains all the definitions needed to use the static database access library. Two structures (`DBBASE` and `DBENTRY`) are used to access a database. The fields in these structures should not be accessed directly. They are used by the static database access library to keep state information for the caller.

14.2 Definitions

14.2.1 DBBASE

Multiple memory resident databases can be accessed simultaneously. The user must provide definitions in the form:

```
DBBASE *pdbname;
```

NOTE: On an IOC `pdbname` is a global variable, which is accessible if you include `dbAccess.h`

14.2.2 DBENTRY

A typical declaration for a database entry structure is:

```
DBENTRY *pdbentry;  
pdbentry=dbAllocEntry(pdbname);
```

Most static access to a database is via a `DBENTRY` structure. As many `DBENTRY`s as desired can be allocated.

The user should NEVER access the fields of `DBENTRY` directly. They are meant to be used by the static database access library.

Most static access routines accept an argument which contains the address of a `DBENTRY`. Each routine uses this structure to locate the information it needs and gives values to as many fields in this structure as possible. All other fields are set to `NULL`.

14.2.3 Field Types

Each database field has a type as defined in the next chapter. For static database access a new and simpler set of field types are defined. In addition, at runtime, a database field can be an array. With static database access, however, all fields are scalars. Static database access field types are called DCT field types.

The DCT field types are:

- **DCT_STRING**: Character string.
- **DCT_INTEGER**: Integer value
- **DCT_REAL**: Floating point number
- **DCT_MENU**: A set of choice strings
- **DCT_MENUFORM**: A set of choice strings with associated form.
- **DCT_INLINK**: Input Link
- **DCT_OUTLINK**: Output Link
- **DCT_FWDLINK**: Forward Link
- **DCT_NOACCESS**: A private field for use by record access routines

A `DCT_STRING` field contains the address of a `NULL` terminated string. The field types `DCT_INTEGER` and `DCT_REAL` are used for numeric fields. A field that has any of these types can be accessed via the `dbGetString`, `dbPutString`, `dbVerify`, and `dbGetRange` routines.

The field type `DCT_MENU` has an associated set of strings defining the choices. Routines are available for accessing menu fields. A menu field can also be accessed via the `dbGetString`, `dbPutString`, `dbVerify`, and `dbGetRange` routines.

The field type `DCT_MENUFORM` is like `DCT_MENU` but in addition the field has an associated link field. The information for the link field can be entered via a set of form manipulation fields.

`DCT_INLINK` (input), `DCT_OUTLINK` (output), and `DCT_FWDLINK` (forward) specify that the field is a link, which has an associated set of static access routines described in the next subsection. A field that has any of these types can also be accessed via the `dbGetString`, `dbPutString`, `dbVerify`, and `dbGetRange` routines.

14.3 Allocating and Freeing DBBASE

14.3.1 dbAllocBase

```
DBBASE *dbAllocBase(void);
```

This routine allocates and initializes a `DBBASE` structure. It does not return if it is unable to allocate storage.

dbAllocBase allocates and initializes a DBBASE structure. Normally an application does not need to call dbAllocBase because a call to dbReadDatabase or dbReadDatabaseFP automatically calls this routine if pdbname is null. Thus the user only has to supply code like the following:

```
DBBASE *pdbname=0;
...
status = dbReadDatabase(&pdbname, "sample.db",
    "<path>", "<macro substitutions>");
```

The static database access library allows applications to work with multiple databases, each referenced via a different (DBBASE *) pointer. Such applications may find it necessary to call dbAllocBase directly.

dbAllocBase does not return if it is unable to allocate storage.

14.3.2 dbFreeBase

```
void dbFreeBase(DBBASE *pdbname);
```

dbFreeBase frees the entire database reference by pdbname including the DBBASE structure itself.

14.4 DBENTRY Routines

14.4.1 Alloc/Free DBENTRY

```
DBENTRY *dbAllocEntry(DBBASE *pdbname);
void dbFreeEntry(DBENTRY *pdbentry);
```

These routines allocate, initialize, and free DBENTRY structures. The user can allocate and free DBENTRY structures as necessary. Each DBENTRY is, however, tied to a particular database.

dbAllocEntry and dbFreeEntry act as a pair, i.e. the user calls dbAllocEntry to create a new DBENTRY and calls dbFreeEntry when done.

14.4.2 dbInitEntry dbFinishEntry

```
void dbInitEntry(DBBASE *pdbname, DBENTRY *pdbentry);
void dbFinishEntry(DBENTRY *pdbentry);
```

The routines dbInitEntry and dbFinishEntry are provided in case the user wants to allocate a DBENTRY structure on the stack. Note that the caller **MUST** call dbFinishEntry before returning from the routine that calls dbInitEntry. An example of how to use these routines is:

```
int xxx(DBBASE *pdbname)
{
    DBENTRY dbentry;
    DBENTRY *pdbentry = &dbentry;
    ...
    dbInitEntry(pdbname, pdbentry);
    ...
    dbFinishEntry(pdbentry);
}
```

14.4.3 dbCopyEntry

dbCopyEntry

Contents

```
DBENTRY *dbCopyEntry(DBENTRY *pdbentry);  
void dbCopyEntryContents(DBENTRY *pfrom,DBENTRY *pto);
```

The routine `dbCopyEntry` allocates a new entry, via a call to `dbAllocEntry`, copies the information from the original entry, and returns the result. The caller must free the entry, via `dbFreeEntry` when finished with the `DBENTRY`.

The routine `dbCopyEntryContents` copies the contents of `pfrom` to `pto`. Code should never perform structure copies.

14.5 Read and Write Database

14.5.1 Read Database File

```
long dbReadDatabase(DBBASE **ppdbbase,const char *filename,  
    char *path, char *substitutions);  
long dbReadDatabaseFP(DBBASE **ppdbbase,FILE *fp,  
    char *path, char *substitutions);  
long dbPath(DBBASE *pdbbase,const char *path);  
long dbAddPath(DBBASE *pdbbase,const char *path);
```

`dbReadDatabase` and `dbReadDatabaseFP` both read a file containing database definitions as described in chapter “Database Definitions”. If `*ppdbbase` is `NULL`, `dbAllocBase` is automatically invoked and the return address assigned to `*ppdbbase`. The only difference between the two routines is that one accepts a file name and the other a “FILE *”. Any combination of these routines can be called multiple times. Each adds definitions with the rules described in chapter “Database Definitions”.

The routines `dbPath` and `dbAddPath` specify paths for use by include statements in database definition files. These are not normally called by user code.

14.5.2 Write Database Definitons

```
long dbWriteMenu(DBBASE *pdbbase,char *filename,char *menuName);  
long dbWriteMenuFP(DBBASE *pdbbase,FILE *fp,char *menuName);  
long dbWriteRecordType(DBBASE *pdbbase,char *filename,char *recordTypeName);  
long dbWriteRecordTypeFP(DBBASE *pdbbase,FILE *fp,char *recordTypeName);  
long dbWriteDevice(DBBASE *pdbbase,char *filename);  
long dbWriteDeviceFP(DBBASE *pdbbase,FILE *fp);  
long dbWriteDriver(DBBASE *pdbbase,char *filename);  
long dbWriteDriverFP(DBBASE *pdbbase,FILE *fp);  
long dbWriteRegistrarFP(DBBASE *pdbbase,FILE *fp);  
long dbWriteFunctionFP(DBBASE *pdbbase,FILE *fp);  
long dbWriteVariableFP(DBBASE *pdbbase,FILE *fp);  
long dbWriteBreaktable(DBBASE *pdbbase,const char *filename);  
long dbWriteBreaktableFP(DBBASE *pdbbase,FILE *fp);
```

Each of these routines writes files in the same format accepted by `dbReadDatabase` and `dbReadDatabaseFP`. Two versions of each type are provided. The only difference is that one accepts a filename and the other a "FILE *". Thus only one of each type has to be described.

`dbWriteMenu` writes the description of the specified menu or, if `menuName` is `NULL`, the descriptions of all menus.

`dbWriteRecordType` writes the description of the specified record type or, if `recordTypeName` is `NULL`, the descriptions of all record types.

`dbWriteDevice` writes the description of all devices to `stdout`.

`dbWriteDriver` writes the description of all drivers to `stdout`.

`dbWriteRegistrarFP` writes the list of all registrars to the given open file (no filename version is provided).

`dbWriteFunctionFP` writes the list of all functions to the given open file (no filename version is provided).

`dbWriteVariableFP` writes the list of all variables to the given open file (no filename version is provided).

14.5.3 Write Record Instances

```
long dbWriteRecord(DBBASE *pdbname, char * file,  
                  char *precordTypeName, int level);  
long dbWriteRecordFP(DBBASE *pdbname, FILE *fp,  
                    char *precordTypeName, int level);
```

Each of these routines writes files in the same format accepted by `dbReadDatabase` and `dbReadDatabaseFP`. Two versions of each type are provided. The only difference is that one accepts a filename and the other a "FILE *". Thus only one of each type has to be described.

`dbWriteRecord` writes record instances. If `precordTypeName` is `NULL`, then the record instances for all record types are written, otherwise only the records for the specified type are written. `level` has the following meaning:

- 0 - Write only prompt fields that are different than the default value.
- 1 - Write only the fields which are prompt fields.
- 2 - Write the values of all fields.

14.6 Manipulating Record Types

14.6.1 Get Number of Record Types

```
int dbGetNRecordTypes(DBENTRY *pdbentry);
```

This routine returns the number of record types in the database.

14.6.2 Locate Record Type

```
long dbFindRecordType(DBENTRY *pdbentry,  
                     char *recordTypeName);  
long dbFirstRecordType(DBENTRY *pdbentry);  
long dbNextRecordType(DBENTRY *pdbentry);
```

`dbFindRecordType` locates a particular record type. `dbFirstRecordType` locates the first, in alphabetical order, record type. Given that `DBENTRY` points to a particular record type, `dbNextRecordType` locates the next record type. Each routine returns 0 for success and a non zero status value for failure. A typical code segment using these routines is:

```
status = dbFirstRecordType(pdbentry);
while(!status) {
    /*Do something*/
    status = dbNextRecordType(pdbentry)
}
```

14.6.3 Get Record Type Name

```
char *dbGetRecordTypeName(DBENTRY *pdbentry);
```

This routine returns the name of the record type that `DBENTRY` currently references. This routine should only be called after a successful call to `dbFindRecordType`, `dbFirstRecordType`, or `dbNextRecordType`. It returns `NULL` if `DBENTRY` does not point to a record description.

14.7 Manipulating Field Descriptions

The routines described in this section all assume that `DBENTRY` references a record type, i.e. that `dbFindRecordType`, `dbFirstRecordType`, or `dbNextRecordType` has returned success or that a record instance has been successfully located.

14.7.1 Get Number of Fields

```
int dbGetNFields(DBENTRY *pdbentry,int dctonly);
```

Returns the number of fields for the record instance that `DBENTRY` currently references.

14.7.2 Locate Field

```
long dbFirstField(DBENTRY *pdbentry,int dctonly);
long dbNextField(DBENTRY *pdbentry,int dctonly);
```

These routines are used to locate fields. If any of these routines returns success, then `DBENTRY` references that field description.

14.7.3 Get Field Type

```
int dbGetFieldType(DBENTRY *pdbentry);
```

This routine returns the integer value for a DCT field type, see Section 14.2.3 on page 194, for a description of the field types.

14.7.4 Get Field Name

```
char *dbGetFieldName(DBENTRY *pdbentry);
```

This routine returns the name of the field that DBENTRY currently references. It returns NULL if DBENTRY does not point to a field.

14.7.5 Get Default Value

```
char *dbGetDefault(DBENTRY *pdbentry);
```

This routine returns the default value for the field that DBENTRY currently references. It returns NULL if DBENTRY does not point to a field or if the default value is NULL.

14.7.6 Get Field Prompt

```
char *dbGetPrompt(DBENTRY *pdbentry);  
int dbGetPromptGroup(DBENTRY *pdbentry);
```

The dbGetPrompt routine returns the character string prompt value, which describes the field. dbGetPromptGroup returns the field group as described in guigroup.h.

14.8 Manipulating Record Attributes

A record attribute is a "psuedo" field definition attached to a record type. If a attribute value is assigned to a psuedo field name then all record instances of that record type appear to have that field with the defined value. All attribute fields are DCT_STRING fields.

Two field attributes are automatically created: RTYP and VERS. RTYP is set equal to ,the record type name. VERS is initialized to the value "none specified" but can be changed by record support.

14.8.1 dbPutRecord Attribute

```
long dbPutRecordAttribute(DBENTRY *pdbentry,  
char *name, char*value)
```

This creates or modifies attribute name with value.

14.8.2 dbGetRecord Attribute

```
long dbGetRecordAttribute(DBENTRY *pdbentry, char *name);
```

14.9 Manipulating Record Instances

With the exception of `dbFindRecord`, each of the routines described in this section require that `DBENTRY` references a valid record type, i.e. that `dbFindRecordType`, `dbFirstRecordType`, or `dbNextRecordType` has been called and returned success.

14.9.1 Get Number of Records

```
int dbGetNRecords(DBENTRY *pdbentry);
```

Returns the number of record instances for the record type that `DBENTRY` currently references.

14.9.2 Locate Record

```
long dbFindRecord(DBENTRY *pdbentry, char *precordName);  
long dbFirstRecord(DBENTRY *pdbentry);  
long dbNextRecord(DBENTRY *pdbentry);
```

These routines are used to locate record instances. If any of these routines returns success, then `DBENTRY` references the record. `dbFindRecord` can be called without `DBENTRY` referencing a valid record type. `dbFirstRecord` only works if `DBENTRY` references a record type. The `dbDumpRecords` example given at the beginning of this chapter shows how these routines can be used.

`dbFindRecord` also calls `dbFindField` if the record name includes a field name, i.e. it ends in “.XXX”. The routine `dbFoundField` returns (`TRUE`, `FALSE`) if the field (was, was not) found. If it was not found, then `dbFindField` must be called before individual fields can be used.

14.9.3 Get Record Name

```
char *dbGetRecordName(DBENTRY *pdbentry);
```

This routine only works properly if called after `dbFindRecord`, `dbFirstRecord`, or `dbNextRecord` has returned success.

14.9.4 Create/Delete/Free Record

```
long dbCreateRecord(DBENTRY *pdbentry, char *precordName);  
long dbDeleteRecord(DBENTRY *pdbentry);  
long dbFreeRecords(DBBASE *pdbname);
```

`dbCreateRecord`, which assumes that `DBENTRY` references a valid record type, creates a new record instance and initializes it as specified by the record description. If it returns success, then `DBENTRY` references the record just created. `dbDeleteRecord` deletes a single record instance/. `dbFreeRecords` deletes all record instances.

14.9.5 Copy Record

```
long dbCopyRecord(DBENTRY *pdbentry, char *newRecordName  
int overwriteOK)
```

This routine copies the record instance currently referenced by DBENTRY. Thus it creates a new record with the name newRecordName that is of the same type as the original record and copies the original records field values to the new record. If newRecordName already exists and overWriteOK is true, then the original newRecordName is deleted and recreated. If dbCopyRecord completes successfully, DBENTRY references the new record.

14.9.6 Rename Record

```
long dbRenameRecord(DBENTRY *pdbentry, char *newname)
```

This routine renames the record instance currently referenced by DBENTRY. If dbRenameRecord completes successfully, DBENTRY references the renamed record.

14.9.7 Record Visibility

These routines are for use by graphical configuration tools.

```
long dbVisibleRecord(DBENTRY *pdbentry);
long dbInvisibleRecord(DBENTRY *pdbentry);
int dbIsVisibleRecord(DBENTRY *pdbentry);
```

dbVisibleRecord sets a record to be visible. dbInvisibleRecord sets a record invisible. dbIsVisibleRecord returns TRUE if a record is visible and FALSE otherwise.

14.9.8 Find Field

```
long dbFindField(DBENTRY *pdbentry, char *pfieldName);
int dbFoundField(DBENTRY *pdbentry);
```

Given that a record instance has been located, dbFindField finds the specified field. If it returns success, then DBENTRY references that field. dbFoundField returns (FALSE, TRUE) if (no field instance is currently available, a field instance is available).

14.9.9 Get/Put Field Values

```
char *dbGetString(DBENTRY *pdbentry);
long dbPutString(DBENTRY *pdbentry, char *pstring);
char *dbVerify(DBENTRY *pdbentry, char *pstring);
char *dbGetRange(DBENTRY *pdbentry);
int dbIsDefaultValue(DBENTRY *pdbentry);
```

These routines are used to get or change field values. They work on all the database field types except DCT_NOACCESS but should **NOT** be used to prompt the user for information for DCT_MENU, DCT_MENUFORM, or DCT_LINK_xxx fields. dbVerify returns (NULL, a message) if the string is (valid, invalid). Please note that the strings returned are volatile, i.e. the next call to a routines that returns a string will overwrite the value returned by a previous call. Thus it is the caller's responsibility to copy the strings if the value must be kept.

DCT_MENU, DCT_MENUFORM and DCT_LINK_xxx fields can be manipulated via routines described in the following sections. If, however dbGetString and dbPutString are used, they do work correctly. For these field types dbGetString and dbPutString are intended to be used only for creating and restoring versions of a database.

14.10 Manipulating Menu Fields

These routines should only be used for DCT_MENU and DCT_MENUFORM fields. Thus they should only be called if dbFindField, dbFirstField, or dbNextField has returned success and the field type is DCT_MENU or DCT_MENUFORM.

14.10.1 Get Number of Menu Choices

```
int dbGetNMenuChoices(DBENTRY *pdbentry);
```

This routine returns the number of menu choices for menu.

14.10.2 Get Menu Choice

```
char **dbGetMenuChoices(DBENTRY *pdbentry);
```

This routine returns the address of an array of pointers to strings which contain the menu choices.

14.10.3 Get/Put Menu

```
int dbGetMenuIndex(DBENTRY *pdbentry);  
long dbPutMenuIndex(DBENTRY *pdbentry, int index);  
char *dbGetMenuStringFromIndex(DBENTRY *pdbentry, int index);  
int dbGetMenuIndexFromString(DBENTRY *pdbentry,  
    char *choice);
```

NOTE: These routines do not work if the current field value contains a macro definition.

dbGetMenuIndex returns the index of the menu choice for the current field, i.e. it specifies which choice to which the field is currently set. dbPutMenuIndex sets the field to the choice specified by the index.

dbGetMenuStringFromIndex returns the string value for a menu index. If the index value is invalid NULL is returned. dbGetMenuIndexFromString returns the index for the given string. If the string is not a valid choice a -1 is returned.

14.10.4 Locate Menu

```
dbMenu *dbFindMenu(DBBASE *pdbname, char *name);
```

dbFindMenu is most useful for runtime use but is a static database access routine. This routine just finds a menu with the given name.

14.11 Manipulating Link Fields

14.11.1 Link Types

Links are the most complicated types of fields. A link can be a constant, reference a field in another record, or can refer to a hardware device. Two additional complications arise for hardware links. The first is that field `DTYP`, which is a menu field, determines if the `INP` or `OUT` field is a device link. The second is that the information that must be specified for a device link is bus dependent. In order to shelter database configuration tools from these complications the following is done for static database access.

- Static database access will treat `DTYP` as a `DCT_MENUFORM` field.
- The information for the link field related to the `DCT_MENUFORM` can be entered via a set of form manipulation routines associated with the `DCT_MENUFORM` field. Thus the link information can be entered via the `DTYP` field rather than the link field.
- The Form routines described in the next section can also be used with any link field.

Each link is one of the following types:

- **DCT_LINK_CONSTANT**: Constant value.
- **DCT_LINK_PV**: A process variable link.
- **DCT_LINK_FORM**: A link that can only be processed via the form routines described in the next chapter.

Database configuration tools can change any link between being a constant and a process variable link. Routines are provided to accomplish these tasks.

The routines `dbGetString`, `dbPutString`, and `dbVerify` can be used for link fields but the form routines can be used to provide a friendlier user interface.

14.11.2 All Link Fields

```
int dbGetNLinks(DBENTRY *pdbentry);
long dbGetLinkField(DBENTRY *pdbentry,int index)
int dbGetLinkType(DBENTRY *pdbentry);
```

These are routines for manipulating `DCT_xxxLINK` fields. `dbGetNLinks` and `dbGetLinkField` are used to walk through all the link fields of a record. `dbGetLinkType` returns one of the values: `DCT_LINK_CONSTANT`, `DCT_LINK_PV`, `DCT_LINK_FORM`, or the value -1 if it is called for an illegal field.

14.11.3 Constant and Process Variable Links

```
long dbCvtLinkToConstant(DBENTRY *pdbentry);
long dbCvtLinkToPvlink(DBENTRY *pdbentry);
```

These routines should be used for modifying `DCT_LINK_CONSTANT` or `DCT_LINK_PV` links. They should not be used for `DCT_LINK_FORM` links, which should be processed via the associated `DCT_MENUFORM` field described above.

14.12 Manipulating MenuForm Fields

These routines are used with a `DCT_MENUFORM` field (a `DTYP` field) to manipulate the associated `DCT_INLINK` or `DCT_OUTLINK` field. They can also be used on any `DCT_INLINK`, `DCT_OUTLINK`, or `DCT_FWDLINK` field.

14.12.1 Alloc/Free Form

```
int dbAllocForm(DBENTRY *pdbentry)
long dbFreeForm(DBENTRY *pdbentry)
```

dbAllocForm allocates storage needed to manipulate forms. The return value is the number of elements in the form. If the current field value contains a macro definition, the number of lines returned is 0.

14.12.2 Get/Put Form

```
char **dbGetFormPrompt(DBENTRY *pdbentry)
char **dbGetFormValue(DBENTRY *pdbentry)
long dbPutForm(DBENTRY *pdbentry, char **value)
```

dbGetFormPrompt returns a pointer to an array of pointers to character strings specifying the prompt string. dbGetFormValue returns the current values. dbPutForm, which can use the same array of values returned by dbGetForm, sets new values.

14.12.3 Verify Form

```
char **dbVerifyForm(DBENTRY *pdbentry, char **value)
```

dbVerifyForm can be called to verify user input. It returns NULL if no errors are present. If errors are present, it returns a pointer to an array of character strings containing error messages. Lines in error have a message and correct lines have a NULL string.

14.12.4 Get Related Field

```
char *dbGetRelatedField(DBENTRY *pdbentry)
```

This routine returns the field name of the related field for a DCT_MENUFORM field. If it is called for any other type of field it returns NULL.

14.12.5 Example

The following is code showing use of these routines:

```
char **value;
char **prompt;
char **error;
int n;

...
n = dbAllocForm(pdbentry);
if(n<=0) {<Error>}
prompt = dbGetFormPrompt(pdbentry);
value = dbGetFormValue(pdbentry);
for(i=0; i<n; i++) {
    printf("%s (%s) : \n",prompt[i],value[i]);
    /*The following accepts input from stdin*/
    scanf("%s",value[i]);
}
error = dbVerifyForm(pdbentry,value);
```

```
if(error) {
    for(i=0; i<n; i++) {
        if(error[i]) printf("Error: %s (%s) %s\n", prompt[i],
            value[i],error[i]);
    }
}else {
    dbPutForm(pdbentry,value)
}
dbFreeForm(pdbentry);
```

All value strings are MAX_STRING_SIZE in length.

A set of form calls for a particular DBENTRY, **MUST** begin with a call to dbAllocForm and end with a call to dbFreeForm. The values returned by dbGetFormPrompt, dbGetFormValue, and dbVerifyForm are valid only between the calls to dbAllocForm and dbFreeForm.

14.13 Manipulating Information Items

Information items are stored as a list attached to each individual record instance. All routines listed in this section require that the DBENTRY argument refer to a valid record instance.

14.13.1 Locate Item

```
long dbFirstInfo(DBENTRY *pdbentry);
long dbNextInfo(DBENTRY *pdbentry);
long dbFindInfo(DBENTRY *pdbentry,const char *name);
```

There are two ways to locate info items, by scanning through the list using first/next, or by asking for the item by name. These routines set pdbentry to refer to the info item and return 0, or return an error code if no info item is found.

14.13.2 Get Item Name

```
const char * dbGetInfoName(DBENTRY *pdbentry);
```

Returns the name of the info item referred to by pdbentry, or a NULL pointer if no item is referred to.

14.13.3 Get/Set Item String Value

```
const char * dbGetInfoString(DBENTRY *pdbentry);
long dbPutInfoString(DBENTRY *pdbentry,const char *string);
```

These routines provide access to the currently selected items' string value. When changing the string value using dbPutInfoString, the character string provided will be copied, with additional memory being allocated as necessary. Users are advised not to make continuously repeated calls to dbPutInfoString at IOC runtime as this could fragment the free memory heap. The Put routine returns 0 if Ok or an error code; the Get routine returns NULL on error.

14.13.4 Get/Set Item Pointer Value

```
void * dbGetInfoPointer(DBENTRY *pdbentry);
long dbPutInfoPointer(DBENTRY *pdbentry, void *pointer);
```

Each info item includes space to store a single `void*` pointer as well as the value string. Applications using the info item may set this as often as they wish. The Put routine returns 0 if Ok or an error code; the Get routine returns NULL on error.

14.13.5 Create/Delete Item

```
long dbPutInfo(DBENTRY *pdbentry, const char *name, const char *string);
long dbDeleteInfo(DBENTRY *pdbentry);
```

A new info item can be created by calling `dbPutInfo`. If an item by that name already exists its value will be replaced with the new string, otherwise storage is allocated and the name and value strings copied into it. The function returns 0 on success, or an error code.

When calling `dbDeleteInfo`, the `pdbentry` must refer to the item to be removed (using `dbFindFirst/dbFindNext` or `dbFindInfo`). The function returns 0 on success, or an error code.

14.13.6 Convenience Routine

```
const char * dbGetInfo(DBENTRY *pdbentry, const char *name);
```

It is common to want to look up the value of a named info item in one call, and `dbGetInfo` is provided for this purpose. It returns a NULL pointer if no info item exists with the given name.

14.14 Find Breakpoint Table

```
brkTable *dbFindBrkTable(DBBASE *pddbbase, char *name)
```

This routine returns the address of the specified breakpoint table. It is normally used by the runtime breakpoint conversion routines so will not be discussed further.

14.15 Dump Routines

```
void dbDumpPath(DBBASE *pddbbase)
void dbDumpRecord(DBBASE *pddbbase, char *precordTypeName, int level);
void dbDumpMenu(DBBASE *pddbbase, char *menuName);
void dbDumpRecordType(DBBASE *pddbbase, char *recordTypeName);
void dbDumpField(DBBASE *pddbbase, char *recordTypeName, char *fname);
void dbDumpDevice(DBBASE *pddbbase, char *recordTypeName);
void dbDumpDriver(DBBASE *pddbbase);
void dbDumpRegistrar(DBBASE *pddbbase);
void dbDumpFunction(DBBASE *pddbbase);
void dbDumpVariable(DBBASE *pddbbase);
void dbDumpBreaktable(DBBASE *pddbbase, char *name);
void dbPvdDump(DBBASE *pddbbase, int verbose);
void dbReportDeviceConfig(DBBASE *pddbbase, FILE *report);
```

These routines are used to dump information about the database. `dbDumpRecord`, `dbDumpMenu`, `dbDumpDriver`, `dbDumpRegistrar` and `dbDumpVariable` just call the corresponding `dbWritexxvFP` routine specifying stdout for the file. `dbDumpRecordType`, `dbDumpField`, and `dbDumpDevice` give internal information useful on an ioc. These commands can be executed via `iocsh`. Just specify `pddbbase` as the first argument.

14.16 Examples

14.16.1 Expand Include

This example is like the `dbExpand` utility, except that it doesn't allow path or macro substitution options. It reads a set of database definition files and writes all definitions to `stdout`. All include statements appearing in the input files are expanded.

```
/* dbExpand.c */
#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>
#include <epicsPrint.h>
#include <dbStaticLib.h>

DBBASE *pdbbase = NULL;

int main(int argc, char **argv)
{
    long    status;
    int     i;
    int     arg;

    if(argc<2) {
        printf("usage: expandInclude file1.db file2.db...\n");
        exit(0);
    }
    for(i=1; i<argc; i++) {
        status = dbReadDatabase(&pdbbase, argv[i], NULL, NULL);
        if(!status) continue;
        fprintf(stderr, "For input file %s", argv[i]);
        errMessage(status, "from dbReadDatabase");
    }
    dbWriteMenuFP(pdbbase, stdout, 0);
    dbWriteRecordTypeFP(pdbbase, stdout, 0);
    dbWriteDeviceFP(pdbbase, stdout);
    dbWriteDriverFP(pdbbase, stdout);
    dbWriteRecordFP(pdbbase, stdout, 0, 0);
    return(0);
}
```

14.16.2 dbDumpRecords

NOTE: This example is similar but not identical to the actual `dbDumpRecords` routine.

The following example demonstrates how to use the database access routines. The example shows how to locate each record and display each field.

```
void dbDumpRecords(DBBASE *pdbbase)
{
```

```
DBENTRY *pdbentry;
long status;

pdbentry = dbAllocEntry(pdbbase);
status = dbFirstRecordType(pdbentry);
if(status) {printf("No record descriptions\n");return;}
while(!status) {
    printf("record type: %s",dbGetRecordTypeName(pdbentry));
    status = dbFirstRecord(pdbentry);
    if(status) printf(" No Records\n");
    else printf("\n Record:%s\n",dbGetRecordName(pdbentry));
    while(!status) {
        status = dbFirstField(pdbentry,TRUE);
        if(status) printf(" No Fields\n");
        while(!status) {
            printf(" %s:%s",dbGetFieldName(pdbentry),
                dbGetString(pdbentry));
            status=dbNextField(pdbentry,TRUE);
        }
        status = dbNextRecord(pdbentry);
    }
    status = dbNextRecordType(pdbentry);
}
printf("End of all Records\n");
dbFreeEntry(pdbentry);
}
```

Chapter 15: Runtime Database Access

15.1 Overview

This chapter describes routines for manipulating and accessing an initialized IOC database.

This chapter is divided into the following sections:

- Database related include files. All of interest are listed and those of general interest are discussed briefly.
- Runtime database access overview.
- Description of each runtime database access routine.
- Runtime modification of link fields.
- Lock Set Routines
- Database to Channel Access Routines
- Old Database Access. This is the interface still used by Channel Access and thus by Channel Access clients.

15.2 Database Include Files

Directory base/`include` contains a number of database related include files. Of interest to this chapter are:

- **dbDefs.h**: Miscellaneous database related definitions
- **dbFldTypes.h**: Field type definitions
- **dbAccess.h**: Runtime database access definitions.
- **link.h**: Definitions for link fields.

15.2.1 dbDefs.h

This file contains a number of database related definitions. The most important are:

- **PVNAME_SZ**: The number of characters allowed in the record name.
- **MAX_STRING_SIZE**: The maximum string size for string fields or menu choices.
- **DB_MAX_CHOICES**: The maximum number of choices for a choice field.

15.2.2 dbFldTypes.h

This file defines the possible field types. A field's type is perhaps its most important attribute. Changing the possible field types is a fundamental change to the IOC software, because many IOC software components are aware of the field types.

The field types are:

- **DBF_STRING**: 40 character ASCII string
- **DBF_CHAR**: Signed character

- **DBF_UCHAR**: Unsigned character
- **DBF_SHORT**: Short integer
- **DBF_USHORT**: Unsigned short integer
- **DBF_LONG**: Long integer
- **DBF_ULONG**: Unsigned long integer
- **DBF_FLOAT**: Floating point number
- **DBF_DOUBLE**: Double precision float
- **DBF_ENUM**: An enumerated field
- **DBF_MENU**: A menu choice field
- **DBF_DEVICE**: A device choice field
- **DBF_INLINK**: Input Link
- **DBF_OUTLINK**: Output Link
- **DBF_FWDLINK**: Forward Link
- **DBF_NOACCESS**: A private field for use by record access routines

A field of type `DBF_STRING`, ..., `DBF_DOUBLE` can be a scalar or an array. A `DBF_STRING` field contains a NULL terminated ascii string. The field types `DBF_CHAR`, ..., `DBF_DOUBLE` correspond to the standard C data types.

`DBF_ENUM` is used for enumerated items, which is analogous to the C language enumeration. An example of an enum field is field `VAL` of a multi bit binary record.

The field types `DBF_ENUM`, `DBF_MENU`, and `DBF_DEVICE` all have an associated set of ASCII strings defining the choices. For a `DBF_ENUM`, the record support module supplies values and thus are not available for static database access. The database access routines locate the choice strings for the other types.

`DBF_INLINK` and `DBF_OUTLINK` specify link fields. A link field can refer to a signal located in a hardware module, to a field located in a database record in the same IOC, or to a field located in a record in another IOC. A `DBF_FWDLINK` can only refer to a record in the same IOC. Link fields are described in a later chapter.

`DBF_INLINK` (input), `DBF_OUTLINK` (output), and `DBF_FWDLINK` (forward) specify that the field is a link structure as defined in `link.h`. There are three classes of links:

1. Constant - The value associated with the field is a floating point value initialized with a constant value. This is somewhat of a misnomer because constant link fields can be modified via `dbPutField` or `dbPutLink`.
2. Hardware links - The link contains a data structure which describes a signal connected to a particular hardware bus. See `link.h` for a description of the bus types currently supported.
3. Process Variable Links - This is one of three types:
 - a. `PV_LINK`: The process variable name.
 - b. `DB_LINK`: A reference to a process variable in the same IOC.
 - c. `CA_LINK`: A reference to a variable located in another IOC.

DCT always creates a `PV_LINK`. When the IOC is initialized each `PV_LINK` is converted either to a `DB_LINK` or a `CA_LINK`.

`DBF_NOACCESS` fields are for private use by record processing routines.

15.2.3 dbAccess.h

This file is the interface definition for the run time database access library, i.e. for the routines described in this chapter.

An important structure defined in this header file is `DBADDR`

```
typedef struct dbAddr{  
    struct dbCommon *precord; /* address of record*/  
}
```

```

void    *pfield;    /* address of field*/
void    *pfldDes;   /* address of struct fldDes*/
void    *asPvt;    /* Access Security Private*/
long    no_elements; /* number of elements (arrays)*/
short   field_type; /* type of database field*/
short   field_size; /* size (bytes) of the field*/
short   special;   /* special processing*/
short   dbr_field_type; /*optimal database request type*/
}DBADDR;

```

- **precord**: Address of record. Note that its type is a pointer to a structure defining the fields common to all record types. The common fields appear at the beginning of each record. A record support module can cast `precord` to point to the specific record type.
- **pfield**: Address of the field within the record. Note that `pfield` provides direct access to the data value.
- **pfldDes**: This points to a structure containing all details concerning the field. See Chapter “Database Structures” for details.
- **asPvt**: A field used by access security.
- **no_elements**: A string or numeric field can be either a scalar or an array. For scalar fields `no_elements` has the value 1. For array fields it is the maximum number of elements that can be stored in the array.
- **field_type**: Field type.
- **field_size**: Size of one element of the field.
- **special**: Some fields require special processing. This specifies the type. Special processing is described later in this manual.
- **dbr_field_type**: This specifies the optimal database request type for this field, i.e. the request type that will require the least CPU overhead.

NOTE: `pfield`, `no_elements`, `field_type`, `field_size`, `special`, and `dbr_field_type` can all be set by record support (`cvt_dbaddr`). Thus `field_type`, `field_size`, and `special` can differ from that specified by `pfldDes`.

15.2.4 link.h

This header file describes the various types of link fields supported by EPICS.

15.3 Runtime Database Access Overview

With the exception of record and device support, all access to the database is via the `channel` or database access routines. Even record support routines access other records only via database or channel access. Channel Access, in turn, accesses the database via database access.

Perhaps the easiest way to describe the database access layer is to list and briefly describe the set of routines that constitute database access. This provides a good look at the facilities provided by the database.

Before describing database access, one caution must be mentioned. The only way to communicate with an IOC database from outside the IOC is via Channel Access. In addition, any special purpose software, i.e. any software not described in this document, should communicate with the database via Channel Access, not database access, even if it resides in the same IOC as the database. Since Channel Access provides network independent access to a database, it must ultimately

call database access routines. The database access interface was changed in 1991, but Channel Access was never changed. Instead a module was written which translates old style database access calls to new. This interface between the old and new style database access calls is discussed in the last section of this chapter.

The database access routines are:

- **dbNameToAddr**: Locate a database variable.
- **dbGetField**: Get values associated with a database variable.
- **dbGetLink**: Get value of field referenced by database link (Macro)
- **dbGetLinkValue**: Get value of field referenced by database link (Subroutine)
- **dbGet**: Routine called by `dbGetLinkValue` and `dbGetField`
- **dbPutField**: Change the value of a database variable.
- **dbPutLink**: Change value referenced by database link (Macro)
- **dbPutLinkValue**: Change value referenced by database link (Subroutine)
- **dbPut**: Routine called by `dbPutxxx` functions.
-
- **putNotifyInit**: Initialize "struct putNotify"
- **putNotifyCleanup**: Cleanup "struct putNotify"
- **dbPutNotify**: A database put with notification on completion
- **dbNotifyCancel**: Cancel `dbPutNotify`
- **dbNotifyAdd**: Add a new record for to notify set.
- **dbNotifyCompletion**: Announce that put notify is complete.
- **dbBufferSize**: Determine number of bytes in request buffer.
- **dbValueSize**: Number of bytes for a value field.
- **dbGetRset**: Get pointer to Record Support Entry Table
- **dbIsValueField**: Is this field the VAL field.
- **dbGetFieldIndex**: Get field index. The first field in a record has index 0.
- **dbGetNelement**: Get number of elements in the field
- **dbIsLinkConnected**: Is the link field connected.
- **dbGetPdbAddrFromLink**: Get address of DBADDR.
- **dbGetLinkDBFtype**: Get field type of link.
- **dbGetControlLimits**: Get Control Limits.
- **dbGetGraphicLimits**: Get Graphic Limits.
- **dbGetAlarmLimits**: Get Alarm Limits
- **dbGetPrecision**: Get Precision
- **dbGetUnits**: Get Units
- **dbGetNelements**: Get Number of Elements
- **dbGetSevr**: Get Severity
- **dbGetTimeStamp**: Get Time Stamp
- **dbPutAttribute** Give a value to a record attribute.
- **dbScanPassive**: Process record if it is passive.
- **dbScanLink**: Process record referenced by link if it is passive.
- **dbProcess**: Process Record

- **dbScanFwdLink**: Scan a forward link.

15.3.1 Database Request Types and Options

Before describing database access structures, it is necessary to describe database request types and request options. When `dbPutField` or `dbGetField` are called one of the arguments is a database request type. This argument has one of the following values:

- **DBR_STRING**: Value is a NULL terminated string
- **DBR_CHAR**: Value is a signed char
- **DBR_UCHAR**: Value is an unsigned char
- **DBR_SHORT**: Value is a short integer
- **DBR_USHORT**: Value is an unsigned short integer
- **DBR_LONG**: Value is a long integer
- **DBR_ULONG**: Value is an unsigned long integer
- **DBR_FLOAT**: Value is an IEEE floating point value
- **DBR_DOUBLE**: Value is an IEEE double precision floating point value
- **DBR_ENUM**: Value is a short which is the enum item
- **DBR_PUT_ACKT**: Value is an unsigned short for setting the ACKT.
- **DBR_PUT_ACKS**: Value is an unsigned short for global alarm acknowledgment.

The request types `DBR_STRING`,..., `DBR_DOUBLE` correspond exactly to valid data types for database fields. `DBR_ENUM` corresponds to database fields that represent a set of choices or options. In particular it corresponds to the fields types `DBF_ENUM`, `DBF_DEVICE`, and `DBF_MENU`. The complete set of database field types are defined in `dbFldTypes.h`. `DBR_PUT_ACKT` and `DBR_PUT_ACKS` are used to perform global alarm acknowledgment.

`dbGetField` also accepts argument options which is a mask containing a bit for each additional type of information the caller desires. The complete set of options is:

- **DBR_STATUS**: returns the alarm status and severity
- **DBR_UNITS**: returns a string specifying the engineering units
- **DBR_PRECISION**: returns a long integer specifying floating point precision.
- **DBR_TIME**: returns the time
- **DBR_ENUM_STRS**: returns an array of strings
- **DBR_GR_LONG**: returns graphics info as long values
- **DBR_GR_DOUBLE**: returns graphics info as double values
- **DBR_CTRL_LONG**: returns control info as long values
- **DBR_CTRL_DOUBLE**: returns control info as double values
- **DBR_AL_LONG**: returns alarm info as long values
- **DBR_AL_DOUBLE**: returns alarm info as double values

15.3.2 Options

Example

The file `dbAccess.h` contains macros for using options. A brief example should show how they are used. The following example defines a buffer to accept an array of up to ten float values. In addition it contains fields for options `DBR_STATUS` and `DBR_TIME`.

```
struct buffer {
    DBRstatus
```

```
    DBRtime
    float  value[10];
} buffer;
```

The associated `dbGetField` call is:

```
long options,number_elements,status;
...
options = DBR_STATUS | DBR_TIME;
number_elements = 10;
status = dbGetField(paddr,DBR_FLOAT,&buffer,&options,&number_elements);
```

Consult `dbAccess.h` for a complete list of macros.

Structure `dbAddr` contains a field `dbr_field_type`. This field is the database request type that most closely matches the database field type. Using this request type will put the smallest load on the IOC.

Channel Access provides routines similar to `dbGetField`, and `dbPutField`. It provides remote access to `dbGetField`, `dbPutField`, and to the database monitors described below.

15.3.3 ACKT and ACKS

The request types `DBR_PUT_ACKT` and `DBR_PUT_ACKS` are used for global alarm acknowledgment. The alarm handler uses these requests. For each of these types the user (normally channel access) passes an unsigned short value. This value represents:

`DBR_PUT_ACKT` - Do transient alarms have to be acknowledged? (0,1) means (no, yes).

`DBR_PUT_ACKS` - The highest alarm severity to acknowledge. If the current alarm severity is less than or equal to this value the alarm is acknowledged.

15.4 Database Access Routines

15.4.1 dbNameToAddr

Locate a process variable, format:

```
long dbNameToAddr(
    char *pname, /*ptr to process variable name */
    struct dbAddr *paddr);
```

The most important goal of database access can be stated simply: Provide quick access to database records and fields within records. The basic rules are:

- Call `dbNameToAddr` once and only once for each field to be accessed.
- Read field values via `dbGetField` and write values via `dbPutField`.

The routines described in this subsection are used by channel access, sequence programs, etc. Record processing routines, however, use the routines described in the next section rather than `dbGetField` and `dbPutField`.

Given a process variable name, this routine locates the process variable and fills in the fields of structure `dbAddr`. The format for a process variable name is:

```
"<record_name>.<field_name>"
```

For example the value field of a record with record name `sample_name` is:

```
“sample_name.VAL”.
```

The record name is case sensitive. Field names always consist of all upper case letters.

`dbNameToAddr` locates a record via a process variable directory (PVD). It fills in a structure (`dbAddr`) describing the field. `dbAddr` contains the address of the record and also the field. Thus other routines can locate the record and field without a search. Although the PVD allows the record to be located via a hash algorithm and the field within a record via a binary search, it still takes about 80 microseconds (25MHz 68040) to located a process variable. Once located the `dbAddr` structure allows the process variable to be accessed directly.

15.4.2 Get Routines

15.4.2.1 dbGetField

Get values associated with a process variable, format:

```
long dbGetField(
    struct dbAddr *paddr,
    short  dbrType, /* DBR_xxx */
    void   *pbuffer, /*addr of returned data */
    long   *options, /*addr of options */
    long   *nRequest, /*addr of number of elements */
    void   *pfl); /*used by monitor routines */
```

Thus routine locks, calls `dbGet`, and unlocks.

15.4.2.2 dbGetLink and dbGetLinkValue

Get value from the field referenced by a database link, format:

```
long dbGetLink(
    struct db_link *pdbLink, /*addr of database link*/
    short  dbrType, /* DBR_xxx*/
    void   *pbuffer, /*addr of returned data*/
    long   *options, /*addr of options*/
    long   *nRequest); /*addr of number of elements desired*/
```

NOTES:

- 1) options can be NULL if no options are desired.
- 2) nRequest can be NULL for a scalar.

`dbGetLink` is actually a macro that calls `dbGetLinkValue`. The macro skips the call for constant links. User code should never call `dbGetLinkValue`.

This routine is called by database access itself and by record support and/or device support routines in order to get values for input links. The value can be obtained directly from other records or via a channel access client. This routine honors the link options (process and maximize severity). In addition it has code that optimizes the case of no options and scalar.

15.4.2.3 dbGet

Get values associated with a process variable, format:

```
long dbGet(
    struct dbAddr *paddr,
    short  dbrType, /* DBR_xxx*/
```

```
void *pbuffer, /*addr of returned data
long *options, /*addr of options*/
long *nRequest, /*addr of number of elements*/
void *pfl); /*used by monitor routines*/
```

Thus routine retrieves the data referenced by `paddr` and converts it to the format specified by `dbrType`.

”options” is a read/write field. Upon entry to `dbGet`, `options` specifies the desired options. When `dbGetField` returns, `options` specifies the options actually honored. If an option is not honored, the corresponding fields in buffer are filled with zeros.

”nRequest” is also a read/write field. Upon entry to `dbGet` it specifies the maximum number of data elements the caller is willing to receive. When `dbGet` returns it equals the actual number of elements returned. It is permissible to request zero elements. This is useful when only option data is desired.

”pfl” is a field used by the Channel Access monitor routines. All other users must set `pfl=NULL`.

`dbGet` calls one of a number of conversion routines in order to convert data from the DBF types to the DBR types. It calls record support routines for special cases such as arrays. For example, if the number of field elements is greater than 1 and record support routine `get_array_info` exists, then it is called. It returns two values: the current number of valid field elements and an offset. The number of valid elements may not match `dbAddr.no_elements`, which is really the maximum number of elements allowed. The offset is for use by records which implement circular buffers.

15.4.3 Put Routines

15.4.3.1 dbPutField

Change the value of a process variable, format:

```
long dbPutField(
    structdbAddr *paddr,
    short dbrType, /* DBR_xxx*/
    void *pbuffer, /*addr of data*/
    long nRequest); /*number of elements to write*/
```

This routine is responsible for accepting data in one of the `DBR_xxx` formats, converting it as necessary, and modifying the database. Similar to `dbGetField`, this routine calls one of a number of conversion routines to do the actual conversion and relies on record support routines to handle arrays and other special cases.

It should be noted that routine `dbPut` does most of the work. The actual algorithm for `dbPutField` is:

1. If the `DISP` field is `TRUE` then, unless it is the `DISP` field itself which is being modified, the field is not written.
2. The record is locked.
3. `dbPut` is called.
4. If the `dbPut` is successful then:
If this is the `PROC` field or if both of the following are `TRUE`: 1) the field is a process passive field, 2) the record is passive.
 - a. If the record is already active ask for the record to be reprocessed when it completes.
 - b. Call `dbScanPassive` after setting `putf` `TRUE` to show the process request came from `dbPutField`.
5. The record is unlocked.

15.4.3.2 dbPutLink and dbPutLinkValue

Change the value referenced by a database link, format:

```
long dbPutLink(
```

```

struct db_link *pdbLink, /*addr of database link*/
short dbrType, /* DBR_XXX*/
void *pbuffer, /*addr of data to write*/
long nRequest); /*number of elements to write*/

```

dbPutLink is actually a macro that calls dbPutLinkValue. The macro skips the call for constant links. User code should never call dbPutLinkValue.

This routine is called by database access itself and by record support and/or device support routines in order to put values into other database records via output links.

For Channel Access links it calls dbCaPutLink.

For database links it performs the following functions:

1. Calls dbPut.
2. Implements maximize severity.
3. If the field being referenced is PROC or if both of the following are true: 1) process_passive is TRUE and 2) the record is passive then:
 - a. If the record is already active because of a dbPutField request then ask for the record to be reprocessed when it completes.
 - b. otherwise call dbScanPassive.

15.4.3.3 dbPut

Put a value to a database field, format:

```

long dbPut(
    struct dbAddr *paddr,
    short dbrType, /* DBR_XXX*/
    void *pbuffer, /*addr of data*/
    long nRequest); /*number of elements to write*/

```

This routine is responsible for accepting data in one of the DBR_XXX formats, converting it as necessary, and modifying the database. Similar to dbGet, this routine calls one of a number of conversion routines to do the actual conversion and relies on record support routines to handle arrays and other special cases.

15.4.4 Put Notify Routines

dbPutNotify is a request to notify the caller when all records that are processed as a result of a put complete processing. Complications occur because of record linking and asynchronous records. A put can cause an entire chain of records to process. If any record is an asynchronous record then record completion means asynchronous completion.

The following rules are implemented:

1. The user code must allocate a putNotify control block and initialize it with a call to putNotifyInit before the first call to dbPutNotify. If a putNotify is already in use, i.e. a dbPutNotify has been issued and the userCallback has not been called, it is illegal to issue a new dbPutNotify with the same putNotify control block. Any such attempt will cause an assert failure.
2. The userCallback routine will be always be called unless dbNotifyCancel is called. The userCallback is called when the dbPutNotify is complete. The user is then free to reuse or delete the putNotify control block. The user supplied callback is called when all processing is complete or when an error is detected. If everything completes synchronously the callback routine will be called BEFORE dbPutNotify returns. The userCallback is called without anything locked
3. If the user calls dbNotifyCancel then the userCallback will NOT be called after dbNotifyCancel returns. It may get called while dbNotifycancel is active. If it is active dbNotifyCancel will not return until userCallback completes.

Thus after `dbNotifyCancel` returns the user may reuse or delete the `putNotify` control block. The `putNotify` control block **MUST NOT** be deleted by `userCallback`.

4. If a `putNotify` is already active on the record associated with the `putNotify`, the new `putNotify` is put on a restart list and automatically restarted.
5. If the record associated with the `putNotify` is already active for some other reason, the `putNotify` takes ownership of the record and starts the put request when the record completes processing.
6. In general a set of records may be processed as a result of a single `dbPutNotify`. If a record in the set is found to be active, either because `PACT` is true or because a `putNotify` already owns the record, then that record is not made part of the set of records that must complete before the `putNotify` request completes.

15.4.4.1 `putNotifyInit`

Initialize a `struct putNotify`. This must be called before the first call to `dbPutNotify`. The user must provide storage for the `struct putNotify` and call `putNotifyInit` to initialize it.

Format:

```
void putNotifyInit(putNotify *pputNotify);
```

15.4.4.2 `putNotifyCleanup`

Cleanup a `struct putNotify`. This should be called before the user frees the storage for a `struct putNotify` or there will be resource leaks.

Format:

```
void putNotifyCleanup(putNotify *pputNotify);
```

15.4.4.3 `dbPutNotify`

Perform a database put and notify when record processing is complete.

Format:

```
void dbPutNotify(putNotify *pputnotify);
```

where `PUTNOTIFY` is

```
typedef enum {
    putNotifyOK,
    putNotifyCanceled,
    putNotifyError
}putNotifyStatus;
```

```
typedef struct putNotify{
    void      (*userCallback)(struct putNotify *);
    DBADDR   *paddr;      /*dbAddr set by dbNameToAddr*/
    void     *pbuffer;    /*address of data*/
    long     nRequest;    /*number of elements to be written*/
    short    dbrType;     /*database request type*/
    void     *usrPvt;     /*for private use of user*/
    /*The following is status of request.Set by dbPutNotify*/
    long     status;
    /*fields private to database access*/
    ...
}putNotify;
```

The caller must allocate a `putNotify` structure and set the fields:

```
userCallback - Routine that is called upon completion
paddr - address of a dbAddr
pbuffer - address of data
nRequest - number of data elements
dbrType - database request type
usrPvt - a void * field that caller can use as needed.
```

The status value in `putNotify.status` is one of

- **putNotifyOK** Success
- **putNotifyCanceled** User issued a `dbNotifyCancel`.
- **putNotifyError** Either `dbPut` or `dbProcess` returned an error.

The user callback is always called unless `dbNotifyCancel` is called before the put notify competes. It may be called while `dbPutNotify` or `dbNotifyCancel` is active.

15.4.4.4 dbNotifyCancel

Cancel an outstanding `dbPutNotify`.

Format:

```
void dbNotifyCancel(putNotify *pputnotify);
```

This cancels an active `dbPutNotify`.

15.4.4.5 dbNotifyAdd

This routine is called by database access itself. It should never be called by user code.

15.4.4.6 dbNotifyCompletion

This routine is called by database access itself. It should never be called by user code.

15.4.5 Utility Routines

15.4.5.1 dbBufferSize

Determine the buffer size for a `dbGetField` request, format:

```
long dbBufferSize(
    short dbrType, /* DBR_****/
    long options, /* options mask*/
    long nRequest); /* number of elements*/
```

This routine returns the number of bytes that will be returned to `dbGetField` if the request type, options, and number of elements are specified as given to `dbBufferSize`. Thus it can be used to allocate storage for buffers.

NOTE: This should become a Channel Access routine

15.4.5.2 dbValueSize

Determine the size a value field, format:

```
dbValueSize(short dbrType); /* DBR_****/
```

This routine returns the number of bytes for each element of type `dbrType`.

NOTE: This should become a Channel Access routine

15.4.5.3 dbGetRset

Get address of a record support entry table.

Format:

```
struct rset *dbGetRset(DBADDR *paddr);
```

This routine returns the address of the record support entry table for the record referenced by the DBADDR.

15.4.5.4 dbIsValueField

Is this field the VAL field of the record?

Format:

```
int dbIsValueField(struct dbFldDes *pdbFldDes);
```

This is the routine that makes the `get_value` record support routine obsolete.

15.4.5.5 dbGetFieldIndex

Get field index.

Format:

```
int dbGetFieldIndex(DBADDR *paddr);
```

Record support routines such as `special` and `cvt_dbaddr` need to know which field the DBADDR references. The include file describing the record contains define statements for each field. `dbGetFieldIndex` returns the index that can be matched against the define statements (normally via a switch statement).

15.4.5.6 dbGetNelements

Get number of elements in a field.

Format:

```
long dbGetNelements(struct link *plink, long *nelements);
```

This sets `*nelements` to the number of elements in the field referenced by `plink`.

15.4.5.7 dbIsLinkConnected

Is the link connected.

Format:

```
int dbIsLinkConnected(struct link *plink);
```

This routine returns (TRUE, FALSE) if the link (is, is not) connected.

15.4.5.8 dbGetPdbAddrFromLink

Get address of DBADDR from link.

Format:

```
DBADDR *dbGetPdbAddrFromLink(struct link *plink);
```

This macro returns the address of the DBADDR for a database link and NULL for all other link types.

15.4.5.9 dbGetLinkDBFtype

Get field type of a link.

Format:

```
int dbGetLinkDBFtype(struct link *plink);
```

15.4.5.10 dbGetControlLimits

Get Control Limits for link.

Format:

```
long dbGetControlLimits(struct link *plink, double *low, double *high);
```

15.4.5.11 dbGetGraphicLimits

Get Graphic Limits for link.

Format:

```
long dbGetGraphicLimits(struct link *plink, double *low, double *high);
```

15.4.5.12 dbGetAlarmLimits

Get Alarm Limits for link.

Format:

```
long dbGetAlarmLimits(struct link *plink,  
double lolo, double *low, double *high, double hihi);
```

15.4.5.13 dbGetPrecision

Get Precision for link.

Format:

```
long dbGetPrecision(struct link *plink, short *precision);
```

15.4.5.14 dbGetUnits

Get Units for link.

Format:

```
long dbGetUnits(struct link *plink, char *units, int unitsSize);
```

15.4.5.15 dbGetSevr

Get Severity for link.

Format:

```
long dbGetSevr(struct link *plink, short *sevr);
```

15.4.5.16 dbGetTimeStamp

Get Time Stamp for record containing link.

Format:

```
long dbGetTimeStamp(struct link *plink, TS_STAMP *pstamp);
```

15.4.6 Attribute Routine

15.4.6.1 dbPutAttribute

Give a value to a record attribute.

```
long dbPutAttribute(char *recordTypename,  
                  char *name, char*value);
```

This sets the record attribute name for record type `recordTypename` to `value`. For example the following would set the version for the ai record.

```
dbPutAttribute("ai", "VERS", "V800.6.95")
```

15.4.7 Process Routines

15.4.7.1 dbScanPassive

dbScanLink

dbScanFwdLink

Process record if it is passive, format:

```
long dbScanPassive(  
    struct dbCommon *pfrom,  
    struct dbCommon *pto); /* addr of record*/  
long dbScanLink(  
    struct dbCommon *pfrom,  
    struct dbCommon *pto);  
void dbScanFwdLink(struct link *plink);
```

`dbScanPassive` and `dbScanLink` are given the record requesting the scan, which may be NULL, and the record to be processed. If the record is passive and `pact=FALSE` then `dbProcess` is called. Note that these routine are called by `dbGetLink`, `dbPutField`, and by `recGblFwdLink`.

`dbScanFwdLink` is given a link that must be a forward link field. It follows the rules for scanning a forward link. That is for `DB_LINKs` it calls `dbScanPassive` and for `CA_LINKs` it does a `dbCaPutLink` if the `PROC` field of record is being addressed.

15.4.7.2 dbProcess

Request that a database record be processed, format:

```
long dbProcess(struct dbCommom *precord);
```

Request that record be processed. Record processing is described in detail below.

15.5 Runtime Link Modification

Database links can be changed at run time but only via a channel access client, i.e. via calls to `dbPutField` but not to `dbPutLink`. The following restrictions apply:

- Only `DBR_STRING` is allowed.
- If a link is being changed to a different hardware link type then the `DTYP` field must be modified before the link field.

- The syntax for the string field is exactly the same as described for link fields in chapter “Database Definition”

NOTE: For this release modification to/from hardware links has not been tested. In addition modification to record/device support will be needed in order to properly support dynamic modification of hardware links.

15.6 Channel Access Monitors

There are facilities within the Channel Access communication infrastructure which allow the value of a process variable to be monitored by a channel access client. It is a responsibility of record support (and db common) to notify the channel access server when the internal state of a process variable has been modified. State changes can include changes in the value of a process variable and also changes in the alarm state of a process variable. The routine “db_post_events()” is called to inform the channel access server that a process variable state change event has occurred.

```
#include <caeventmask.h>

int db_post_events(void *precord, void *pfield,
    unsigned intselect);
```

The first argument, “precord”, should be passed a pointer to the record which is posting the event(s). The second argument, “pfield”, should be passed a pointer to the field in the record that contains the process variable that has been modified. The third argument, “select”, should be passed an event select mask. This mask can be any logical or combination of {DBE_VALUE, DBE_LOG, DBE_ALARM}. A description of the purpose of each flag in the event select mask follows.

- DBE_VALUE This indicates that a significant change in the process variable’s value has occurred. A significant change is often determined by the magnitude of the monitor “dead band” field in the record.
- DBE_LOG This indicates that a change in the process variable’s value significant to archival clients has occurred. A significant change to archival clients is often determined by the magnitude of the archive “dead band” field in the record.
- DBE_ALARM This indicates that a change in the process variable’s alarm state has occurred.

The function “db_post_events()” returns 0 if it is successful and -1 if it fails. It appears to be common practice within EPICS record support to ignore the status from “db_post_events()”. At this time “db_post_events()” always returns 0 (success). because so many records at this time depend on this behavior it is unlikely that it will be changed in the future.

The function “db_post_events()” is written so that record support will never be blocked attempting to post an event because a slow client is not able to process events fast enough. Each call to “db_post_events()” causes the current value, alarm status, and time stamp for the field to be copied into a ring buffer. The thread calling “db_post_events()” will not be delayed by any network or memory allocation overhead. A lower priority thread in the server is responsible for transferring the events in the event queue to the channel access clients that may be monitoring the process variable.

Currently, when an event is posted for a DBF_STRING field or a field containing array data the value is NOT saved in the ring buffer and the client will receive whatever value happens to be in the field when the lower priority thread transfers the event to the client. This behavior may be improved in the future.

15.7 Lock Set Routines

User code only calls dbScanLock and dbScanUnlock. All other routines are called by iocCore.

15.7.0.1 dbScanLock

Lock a lock set:

```
long void dbScanLock(struct dbCommon *precord);
```

Lock the lock set to which the specified record belongs.

15.7.0.2 dbScanUnlock

Unlock a lock set:

```
long void dbScanUnlock(struct dbCommon *precord);
```

Lock the lock set to which the specified record belongs

15.7.0.3 dbLockGetLockId

Get lock set id:

```
long dbLockGetLockId(struct dbCommon *precord);
```

Each lock set is assigned a unique ID. This routine retrieves it. This is most useful to determine if two records are in the same lock set.

15.7.0.4 dbLockInitRecords

Determine lock sets for each record in database.

```
void dbLockInitRecords(dbBase *pdbbase);
```

Called by iocInit.

15.7.0.5 dbLockSetMerge

Merge records into same lock set.

```
void dbLockSetMerge(struct dbCommon *pfirst,  
                    struct dbCommon *psecond);
```

If specified records are not in same lock set the lock sets are merged. Called by dbLockInitRecords and also when links are modified by dbPutField.

15.7.0.6 dbLockSetSplitSl

Recompute lock sets for given lock set

```
void dbLockSetSplit(struct dbCommon *psource);
```

This is called when dbPutField modifies links.

15.7.0.7 dbLockSetGblLock

Global lock for modifying links.

```
void dbLockSetGblLock(void);
```

Only one task at a time can modify link fields. This routine provides a global lock to prevent conflicts.

15.7.0.8 dbLockSetGblUnlock

Unlock the global lock.

```
void dbLockSetGblUnlock(void);
```

15.7.0.9 dbLockSetRecordLock

If record is not already scan locked lock it.

```
void dbLockSetRecordLock(struct dbCommon *precord);
```

15.8 Channel Access Database Links

The routines described here are used to create and manipulate Channel Access connections from database input or output links. At IOC initialization an attempt is made to convert all process variable links to database links. For any link that fails, it is assumed that the link is a Channel Access link, i.e. a link to a process variable defined in another IOC. The routines described here are used to manage these links. User code never needs to call these routines. They are automatically called by `iocInit` and database access.

At `iocInit` time a task `dbCaLink` is spawned. This task is a channel access client that issues channel access requests for all channel access links in the database. For each link a channel access search request is issued. When the search succeeds a channel access monitor is established. The monitor is issued specifying `ca_field_type` and `ca_element_count`. A buffer is also allocated to hold monitor return data as well as severity. When `dbCaGetLink` is called data is taken from the buffer, converted if necessary, and placed in the location specified by the `pbuffer` argument.

When the first `dbCaPutLink` is called for a link an output buffer is allocated, again using `ca_field_type` and `ca_element_count`. The data specified by the `pbuffer` argument is converted and stored in the buffer. A request is then made to `dbCaLink` task to issue a `ca_put`. Subsequent calls to `dbCaPutLink` reuse the same buffer.

15.8.1 Basic Routines

Except for `dbCaPutLinkCallback`, these routines are normally only called by database access, i.e. they are not called by record support modules.

15.8.1.1 dbCaLinkInit

Called by `iocInit` to initialize the `dbCa` library

```
void dbCaLinkInit(void);
```

15.8.1.2 dbCaAddLink

Add a new channel access link

```
void dbCaAddLink(struct link *plink);
```

15.8.1.3 dbCaAddLinkCallback

```
void dbCaAddLinkCallback(struct link *plink,  
                          dbCaCallback connect,dbCaCallback monitor,void *userPvt);
```

`connect` will be called whenever the link connects or disconnects. `monitor` will be called whenever a monitor event occurs. `connect` and or `monitor` may be null.

15.8.1.4 dbCaRemoveLink

Remove channel access link.

```
void dbCaRemoveLink(struct link *plink);
```

15.8.1.5 dbCaGetLink

Get link value

```
long dbCaGetLink(struct link *plink,short dbrType,  
void *pbuffer,unsigned short *psevr,long *nRequest);
```

15.8.1.6 dbCaPutLink

Put link value

```
long dbCaPutLink(struct link *plink,short dbrType,  
void *buffering nRequest);
```

15.8.1.7 dbCaPutLinkCallback

This is meant for use by device or record support that wants a put to complete before completing record processing.

```
long dbCaPutLinkCallback(struct link *plink,short dbrType,  
const void *pbuffer,long nRequest,dbCaPutCallback callback);
```

<base>/src/dev/devSoft/devAoSoftCallback.c provides an example of how to use this function. It contains:

```
static long write_ao(aoRecord *pao)  
{  
    struct link *plink = &pao->out;  
    long status;  
  
    if(pao->pact) return(0);  
    if(plink->type!=CA_LINK) {  
        status = dbPutLink(&pao->out,DBR_DOUBLE,&pao->oval,1);  
        return(status);  
    }  
    status = dbCaPutLinkCallback(plink,DBR_DOUBLE,&pao->oval,1,  
        (dbCaCallback)dbCaCallbackProcess,plink);  
    if(status) {  
        recGblSetSevr(pao,LINK_ALARM,INVALID_ALARM);  
        return(status);  
    }  
    pao->pact = TRUE;  
    return(0);  
}
```

What happens is the following:

When the record is processed write_ao is called with pact=0.

If the link is not a CA_LINK it just calls dbPutLink. It leaves pact 0. Thus record support completes.

If it is a CA_LINK it calls dbCaPutLinkCallback and sets pact true. Thus record is asynchronous.

If the record is asynchronous then sometime later dbCaCallbackProcess is called. It calls the process routine of record support, which calls write_ao with pact true. write_ao just returns success. Record support then completes the second phase of record processing.

There is a possibility that the link is changed between the two phases of record processing. If this happens the user supplied callback will still get called exactly once but the link may have been modified.

15.8.2 Attributes of Link

The routines in this section are meant for use by device support to find out information about link fields. They must be called with dbScanLock held, i.e. normally they are called by the read or write method provided by device support.

15.8.2.1 dbCaIsLinkConnected

Is Channel Connected

```
int dbCaIsLinkConnected(struct link *plink)
```

This routine returns (TRUE, FALSE) if the link (is, is not) connected.

15.8.2.2 dbCaGetNelements

Get Number of Elements

```
long dbCaGetNelements(struct link *plink, long *nelements);
```

This call, which returns an error if the link is not connected, sets the native number of elements.

15.8.2.3 dbCaGetSevr

Get Alarm Severity

```
long dbCaGetSevr(struct link *plink, short *severity);
```

This call, which returns an error if the link is not connected, sets the alarm severity.

15.8.2.4 dbCaGetTimeStamp

Get Time Stamp

```
long dbCaGetTimeStamp(struct link *plink, TS_STAMP *pstamp);
```

This call, which returns an error if the link is not connected, sets pstamp to the time obtained by the last CA monitor.

15.8.2.5 dbCaGetLinkDBFtype

Get link type

```
int dbCaGetLinkDBFtype(struct link *plink);
```

This call, which returns an error if the link is not connected, returns the field type.

15.8.2.6 dbCaGetAttributes

Get Attributes

```
long dbCaGetAttributes(struct link *plink,  
void (*callback)(void *usrPvt), void *usrPvt);
```

When ever dbCa receives a connection it issues a CA get request to obtain the control, graphic, and alarm limits and to obtain the precision and units. By calling dbCaGetAttributes the caller can be notified when this get completes.

15.8.2.7 dbCaGetControlLimits

Get Control Limits

```
long dbCaGetControlLimits(struct link *plink, double *low, double *high);
```

This call returns an error if the link is not connected or if the CA get request for limits, etc has not completed. If it returns success it has set the control limits.

15.8.2.8 dbCaGetGraphicLimits

Get graphic Limits

```
long dbCaGetGraphicLimits(struct link *plink, double *low, double *high);
```

This call returns an error if the link is not connected or if the CA get request for limits, etc has not completed. If it returns success it has set the graphic limits.

15.8.2.9 dbCaGetAlarmLimits

Get Alarm Limits

```
long dbCaGetAlarmLimits(struct link *plink,  
double *lolo, double *low, double *high, double *hihi);
```

This call returns an error if the link is not connected or if the CA get request for limits, etc has not completed. If it returns success it has set the alarm limits.

15.8.2.10 dbCaGetPrecision

Get Precision

```
long dbCaGetPrecision(struct link *plink, short *precision);
```

This call returns an error if the link is not connected or if the CA get request for limits, etc has not completed. If it returns success it has set the precision.

15.8.2.11 dbCaGetUnits

Get Units

```
long dbCaGetUnits(struct link *plink, char *units, int unitsSize);
```

This call returns an error if the link is not connected or if the CA get request for limits, etc has not completed. If it returns success it has set the units.

Chapter 16: EPICS General Purpose Tasks

16.1 Overview

This chapter describes two sets of EPICS supplied general purpose tasks: 1) Callback, and 2) Task Watchdog.

Often when writing code for an IOC there is no obvious task under which to execute. A good example is completion code for an asynchronous device support module. EPICS supplies the callback tasks for such code.

If an IOC tasks "crashes" there is normally no one monitoring the vxWorks shell to detect the problem. EPICS provides a task watchdog task which periodically checks the state of other tasks. If it finds that a monitored task has terminated or suspended it issues an error message and can also call other routines which can take additional actions. For example a subroutine record can arrange to be put into alarm if a monitored task crashes.

Since IOCs normally run autonomously, i.e. no one is monitoring the vxWorks shell, IOC code that issues `printf` calls generates errors messages that are never seen. In addition the vxWorks implementation of `fprintf` requires much more stack space than `printf` calls. Another problem with vxWorks is the `logMsg` facility. `logMsg` generates messages at higher priority than all other tasks except the shell. EPICS solves all of these problems via an error message handling facility. Code can call any of the routines `errMessage`, `errPrintf`, or `errlogPrintf`. Any of these result in the error message being generated by a separate low priority task. The calling task has to wait until the message is handled but other tasks are not delayed. In addition the message can be sent to a system wide error message file.

16.2 General Purpose Callback Tasks

16.2.1 Overview

EPICS provides three general purpose IOC callback tasks. The only difference between the tasks is scheduling priority: Low, Medium, and High. The low priority task runs at a priority just higher than Channel Access, the medium at a priority about equal to the median of the periodic scan tasks, and the high at a priority higher than the event scan task. The callback tasks provide a service for any software component that needs a task under which to run. The callback tasks use the task watchdog (described below). They use a rather generous stack and can thus be used for invoking record processing. For example the I/O event scanner uses the general purpose callback tasks.

The following steps must be taken in order to use the general purpose callback tasks:

1. Include callback definitions:

```
#include <callback.h>
```

2. Provide storage for a structure that is a private structure for the callback tasks:

```
CALLBACK mycallback;
```

It is permissible for this to be part of a larger structure, e.g.

```
struct {  
    ...  
    CALLBACK mycallback;  
    ...  
} ...
```

3. Call routines (actually macros) to initialize fields in CALLBACK:

```
callbackSetCallback(VOIDFUNCPTR, CALLBACK *);
```

This defines the callers callback routine. The first argument is the address of a function returning VOID. The second argument is the address of the CALLBACK structure.

```
callbackSetPriority(int, CALLBACK *);
```

The first argument is the priority, which can have one of the values: `priorityLow`, `priorityMedium`, or `priorityHigh`. These values are defined in `callback.h`. The second argument is again the address of the CALLBACK structure.

```
callbackSetUser(void *, CALLBACK *);
```

This call is used to save a value that can be retrieved via a call to:

```
callbackGetUser(void *, CALLBACK *);
```

4. Whenever a callback request is desired just call one of the following:

```
callbackRequest(CALLBACK *);  
callbackRequestProcessCallback(CALLBACK *, int, void *);
```

Either can be called from interrupt level code. The callback routine is passed a single argument, which is the same argument that was passed to `callbackRequest`, i.e., the address of the CALLBACK structure.

16.2.2 Syntax

The following calls are provided:

```
void callbackInit(void);
```

```
void callbackSetCallback(void *pcallbackFunction,  
    CALLBACK *pcallback);  
void callbackSetPriority(int priority, CALLBACK *pcallback);  
void callbackSetUser(void *user, CALLBACK *pcallback);  
void callbackGetUser(void *user, CALLBACK *pcallback);
```

```
void callbackRequest(CALLBACK *);  
void callbackRequestProcessCallback(CALLBACK *pCallback,  
    int Priority, void *pRec);  
void callbackRequestDelayed(CALLBACK *pCallback, double seconds);  
void callbackRequestProcessCallbackDelayed(
```

```
CALLBACK *pCallback, int Priority, void *pRec, double seconds);
int callbackSetQueueSize(int size);
```

Notes:

- callbackInit is performed automatically when EPICS initializes and IOC. Thus user code never calls this function.
- callbackSetCallback, callbackSetPriority, callbackSetUser, and callbackGetUser are actually macros.
- Both callbackRequest and callbackRequestProcessCallback can be called from interrupt handler routines.
- The delayed version of the callbackRequest routines wait the specified amount of time before scheduling the callback.
- callbackRequestProcessCallback is designed for the completion phase of asynchronous record processing. It issues the calls:

```
callbackSetCallback(ProcessCallback, pCallback);
callbackSetPriority(Priority, pCallback);
callbackSetUser(pRec, pCallback);
callbackRequest(pCallback);
```

ProcessCallback, which is designed for asynchronous device completion applications, consists of the following code:

```
static void ProcessCallback(CALLBACK *pCallback)
{
    dbCommon    *pRec;
    struct rset *prset;

    callbackGetUser(pRec, pCallback);
    prset = (struct rset *)pRec->rset;
    dbScanLock(pRec);
    (*prset->process)(pRec);
    dbScanUnlock(pRec);
}
```

16.2.3 Example

An example use of the callback tasks.

```
#include <callback.h>

static structure {
    char    begid[80];
    CALLBACK callback;
    char    endid[80];
}myStruct;

void myCallback(CALLBACK *pcallback)
{
    struct myStruct *pmyStruct;
    callbackGetUser(pmyStruct, pcallback)
```

```
    printf("begid=%s endid=%s\n",&myStruct->begid[0],
          &myStruct->endid[0]);
}
example(char *pbegid, char*pendid)
{
    strcpy(&myStruct.begid[0],pbegid);
    strcpy(&myStruct.endid[0],pendid);
    callbackSetCallback(myCallback,&myStruct.callback);
    callbackSetPriority(priorityLow,&myStruct.callback);
    callbackSetUser(&myStruct,&myStruct.callback);
    callbackRequest(&myStruct.callback);
}
```

The example can be tested by issuing the following command to the vxWorks shell:

```
example("begin","end")
```

This simple example shows how to use the callback tasks with your own structure that contains the CALLBACK structure at an arbitrary location.

16.2.4 Callback Queue

The callback requests put the requests on a vxWorks ring buffer. Thus buffer is set by default to hold 2000 requests. This value can be changed by calling `callbackSetQueueSize` before `incInit` in the startup file. The syntax is:

```
int callbackSetQueueSize(int size)
```

16.3 Task Watchdog

EPICS provides an IOC task that is a watchdog for other tasks. Any task can make a request to be watched. The task watchdog runs periodically and checks each task in its task list. If any task is suspended, an error message is issued and, optionally, a callback task is invoked. The task watchdog provides the following features:

1. Include module:

```
#include <taskwd.h>
```

2. Insert request:

```
taskwdInsert (int tid, TASKWDFUNCPRR callback,
             VOID *userarg);
```

This is the request to include the task with the specified `tid` in the list of tasks to be watched. If `callback` is not `NULL` then if the task becomes suspended, the callback routine will be called with a single argument `userarg`.

3. Remove request:

```
taskwdRemove(int tid);
```

This routine would typically be called from the callback routine invoked when the original task goes into the suspended state.

4. Insert request to be notified if any task suspends:

```
taskwdAnyInsert(void *userpvt, TASKWDFUNCPRR callback,  
                VOID *userarg);
```

The callback routine will be called whenever any of the tasks being monitored by the task watchdog task suspends. `userpvt` must have a non NULL unique value `taskwdAnyInsert`, because the task watchdog system uses this value to determine who to remove if `taskwdAnyRemove` is called.

5. Remove request for `taskwdAnyInsert` :

```
taskwdAnyRemove(void *userpvt);
```

`userpvt` is the value that was passed to `taskwdAnyInsert`.

Chapter 17: Database Scanning

17.1 Overview

Database scanning is the mechanism for deciding when to process a record. Five types of scanning are possible:

- **Periodic:** A record can be processed periodically. A number of time intervals are supported.
- **Event:** Event scanning is based on the posting of an event by another component of the software via a call to the routine `post_event`.
- **I/O Event:** The original meaning of this scan type is a request for record processing as a result of a hardware interrupt. The mechanism supports hardware interrupts as well as software generated events.
- **Passive:** Passive records are processed only via requests to `dbScanPassive`. This happens when database links (Forward, Input, or Output), which have been declared "Process Passive" are accessed during record processing. It can also happen as a result of `dbPutField` being called (This normally results from a Channel Access put request).
- **Scan Once:** In order to provide for caching puts, The scanning system provides a routine `scanOnce` which arranges for a record to be processed one time.

This chapter explains database scanning in increasing order of detail. It first explains database fields involved with scanning. It next discusses the interface to the scanning system. The last section gives a brief overview of how the scanners are implemented.

17.2 Scan Related Database Fields

The following fields are normally defined via DCT. It should be noted, however, that it is quite permissible to change any of the scan related fields of a record dynamically. For example, a display manager screen could tie a menu control to the SCAN field of a record and allow the operator to dynamically change the scan mechanism.

17.2.1 SCAN

This field, which specifies the scan mechanism, has an associated menu of the following form:

- Passive:** Passively scanned.
- Event:** Event Scanned. The field `EVNT` specifies event number
- I/O Event scanned.**
- 10 Second:** Periodically scanned - Every 10 seconds
- ...
- .1 Second:** Periodically scanned - Every .1 seconds

17.2.2 PHAS

This field determines processing order for records that are in the same scan set. For example all records periodically scanned at a 2 second rate are in the same scan set. All Event scanned records with the same EVNT are in the same scan set, etc. For records in the same scan set, all records with PHAS=0 are processed before records with PHAS=1, which are processed before all records with PHAS=2, etc.

In general it is not a good idea to rely on PHAS to enforce processing order. It is better to use database links.

17.2.3 EVNT - Event Number

This field only has meaning when SCAN is set to Event scanning, in which case it specifies the event number. In order for a record to be event scanned, EVNT must be in the range 0,...255. It should also be noted that some EPICS software components will not request event scanning for event 0. One example is the eventRecord record support module. Thus the application developer will normally want to define events in the range 1,...,255.

17.2.4 PRIO - Scheduling Priority

This field can be used by any software component that needs to specify scheduling priority, e.g. the event and I/O event scan facility uses this field.

17.3 Scan Related Software Components

17.3.1 menuScan.dbd

This file contains definitions for a menu related to field SCAN. The definitions are of the form:

```
menu(menuScan) {  
    choice(menuScanPassive, "Passive")  
    choice(menuScanEvent, "Event")  
    choice(menuScanI_O_Intr, "I/O Intr")  
    choice(menuScan10_second, "10 second")  
    choice(menuScan5_second, "5 second")  
    choice(menuScan2_second, "2 second")  
    choice(menuScan1_second, "1 second")  
    choice(menuScan_5_second, ".5 second")  
    choice(menuScan_2_second, ".2 second")  
    choice(menuScan_1_second, ".1 second")  
}
```

The first three choices must appear first and in the order shown. The remaining definitions are for the periodic scan rates, which must appear in order of decreasing rate. At IOC initialization, the menu values are read by scan initialization. The number of periodic scan rates and the value of each rate is determined from the menu values. Thus periodic scan rates can be changed by changing menuScan.dbd and loading this version via dbLoadDatabase. The only requirement is that each periodic definition must begin with the value and the value must be in units of seconds.

17.3.2 dbScan.h

All software components that interact with the scanning system must include this file.

The most important definitions in this file are:

```
/* Note that these must match the first four definitions*/
/* in choiceGbl.dbd*/
#define SCAN_PASSIVE    0
#define SCAN_EVENT     1
#define SCAN_IO_EVENT   2
#define SCAN_1ST_PERIODIC 3

/*definitions for SCAN_IO_EVENT */
typedef void * IOSCANPVT;
extern int interruptAccept;

long scanInit(void);
void post_event(int event);
void scanAdd(struct dbCommon *);
void scanDelete(struct dbCommon *);
double scanPeriod(int scan);
void scanOnce(void *precord);
int scanOnceSetQueueSize(int size);
int scanppl(void); /*print periodic lists*/
int scanpel(void); /*print event lists*/
int scanpiol(void); /*print io_event list*/
void scanIoInit(IOSCANPVT *);
void scanIoRequest(IOSCANPVT);
```

The first set of definitions defines the various scan types. The next two definitions (`IOSCANPVT` and `interruptAccept`) are for interfacing with the I/O event scanner. The remaining definitions define the public scan access routines. These are described in the following subsections.

17.3.3 Initializing Database Scanners

```
scanInit(void);
```

The routine `scanInit` is called by `iocInit`. It initializes the scanning system.

17.3.4 Adding And Deleting Records From Scan List

The following routines are called each time a record is added or deleted from a scan list.

```
scanAdd(struct dbCommon *);
scanDelete(struct dbCommon *);
```

These routines are called by `scanInit` at IOC initialization time in order to enter all records created via DCT into the correct scan list. The routine `dbPut` calls `scanDelete` and `scanAdd` each time a scan related field is changed (each scan related field is declared to be `SPC_SCAN` in `dbCommon.dbd`). `scanDelete` is called before the field is modified and `scanAdd` after the field is modified.

17.3.5 Obtaining the scan period from the SCAN field

```
double scanPeriod(int scan);
```

The argument is an offset into the set of enum choices for menuScan.h. Most users will just use the SCAN field of a database record. It returns the scan period in seconds. The result will be 0.0 if scan doesn't refer to a periodic rate.

17.3.6 Declaring Database Event

Whenever any software component wants to declare a database event, it just calls:

```
post_event(event)
```

This can be called by virtually any IOC software component. For example sequence programs can call it. The record support module for eventRecord calls it.

17.3.7 Interfacing to I/O Event Scanning

Interfacing to the I/O event scanner is done via some combination of device and driver support.

1. Include <dbScan.h>
2. For each separate event source the following must be done:
 - a. Declare an IOSCANPVT variable, e.g.

```
static IOSCANPVT ioscanpvt;
```
 - b. Call scanIoInit, e.g.

```
scanIoInit(&ioscanpvt);
```
3. Provide the device support get_ioint_info routine. This routine has the format:

```
long get_ioint_info(  
    int cmd,  
    struct dbCommon *precord,  
    IOSCANPVT *ppvt);
```

This routine is called each time the record pointed to by precord is added or deleted from an I/O event scan list. cmd has the value (0,1) if the record is being (added to, deleted from) an I/O event list. This routine must give a value to *ppvt.

4. Whenever an I/O event is detected call scanIoRequest, e.g.

```
scanIoRequest(ioscanpvt)
```

This routine can be called from interrupt level. The request is actually directed to one of the standard callback tasks. The actual one is determined by the PRIO field of dbCommon.

The following code fragment shows an event record device support module that supports I/O event scanning:

```
#include <vxWorks.h>  
#include <types.h>  
#include <stdioLib.h>  
#include <intLib.h>  
#include <dbDefs.h>  
#include <dbAccess.h>  
#include <dbScan.h>  
#include <recSup.h>  
#include <devSup.h>  
#include <eventRecord.h>  
/* Create the dset for devEventXXX */
```

```
long init();
long get_ioint_info();
struct {
    long number;
    DEVSUPFUN  report;
    DEVSUPFUN  init;
    DEVSUPFUN  init_record;
    DEVSUPFUN  get_ioint_info;
    DEVSUPFUN  read_event;
}devEventTestIoEvent={
    5,
    NULL,
    init,
    NULL,
    get_ioint_info,
    NULL};
static IOSCANPVT ioscanpvt;
static void int_service(IOSCANPVT ioscanpvt)
{
    scanIoRequest(ioscanpvt);
}

static long init()
{
    scanIoInit(&ioscanpvt);
    intConnect(<vector>, (FUNCPTR)int_service, ioscanpvt);
    return(0);
}
static long get_ioint_info(
    int cmd,
    struct eventRecord *pr,
    IOSCANPVT *ppvt)
{
    *ppvt = ioscanpvt;
    return(0);
}
```

17.4 Implementation Overview

The code for the entire scanning system resides in `dbScan.c`, i.e. periodic, event, and I/O event. This section gives an overview of how the code in `dbScan.c` is organized. The listing of `dbScan.c` must be studied for a complete understanding of how the scanning system works.

17.4.1 Definitions And Routines Common To All Scan Types

Everything is built around two basic structures:

```
struct scan_list {
    FAST_LOCK lock;
    ELLLIST list;
```

```
    short   modified;
    long   ticks; /*used only for periodic scan sets*/
};

struct scan_element{
    ELLNODE node;
    struct scan_list *pscan_list;
    struct dbCommon *precord;
}
```

Later we will see how `scan_lists` are determined. For now just realize that `scan_list.list` is the head of a list of records that belong to the same scan set (for example, all records that are periodically scanned at a 1 second rate are in the same scan set). The node field in `scan_element` contain the list links. The normal `vxWorks lstLib` routines are used to access the list. Each record that appears in some scan list has an associated `scan_element`. The `SPVT` field which appears in `dbCommon` holds the address of the associated `scan_element`.

The `lock`, `modified`, and `pscan_list` fields allow `scan_elements`, i.e. records, to be dynamically removed and added to scan lists. If `scanList`, the routine which actually processes a scan list, is studied it can be seen that these fields allow the list to be scanned very efficiently if no modifications are made to the list while it is being scanned. This is, of course, the normal case.

The `dbScan.c` module contains several private routines. The following access a single scan set:

- **printList:** Prints the names of all records in a scan set.
- **scanList:** This routine is the heart of the scanning system. For each record in a scan set it does the following:

```
dbScanLock(precord);
dbProcess(precord);
dbScanUnlock(precord);
```

It also has code to recognize when a scan list is modified while the scan set is being processed.
- **addToList:** This routine adds a new element to a scan list.
- **deleteFromList:** This routine deletes an element from a scan list.

17.4.2 Event Scanning

Event scanning is built around the following definitions:

```
#define MAX_EVENTS 256
typedef struct event_scan_list {
    CALLBACK      callback;
    scan_list     scan_list;
}event_scan_list;
static event_scan_list
    *pevent_list[NUM_CALLBACK_PRIORITIES][MAX_EVENTS];
```

`pevent_list` is a 2d array of pointers to `scan_lists`. Note that the array allows for 256 events, i.e. one for each possible event number. In other words, each event number and priority has its own scan list. No `scan_list` is actually created until the first request to add an element for that event number. The event scan lists have the memory layout illustrated in Figure 17-1.

17.4.2.1 post_event

```
post_event(int event)
```

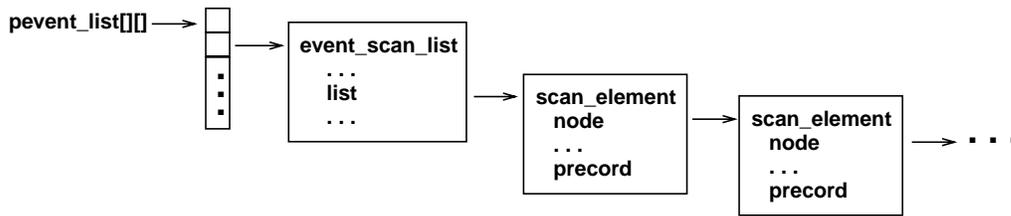


Figure 17-1: Scan List Memory Layout

This routine is called to request event scanning. It can be called from interrupt level. It looks at each `event_scan_list` referenced by `pevent_list[*][event]` (one for each callback priority) and if any elements are present in the `scan_list` a `callbackRequest` is issued. The appropriate callback task calls routine `eventCallback`, which just calls `scanList`.

17.4.3 I/O Event Scanning

I/O event scanning is built around the following definitions:

```

struct io_scan_list {
    CALLBACK    callback;
    struct scan_list    scan_list;
    struct io_scan_list    *next;
}
static struct io_scan_list
    *iosl_head[NUM_CALLBACK_PRIORITIES]
    = {NULL, NULL, NULL};
  
```

The array `iosl_head` and the field `next` are only kept so that `scanpio1` can be implemented and will not be discussed further. I/O event scanning uses the general purpose callback tasks to perform record processing, i.e. no task is spawned for I/O event. The `callback` field of `io_scan_list` is used to communicate with the callback tasks.

The following routines implement I/O event scanning:

17.4.3.1 scanIoInit

```
scanIoInit (IOSCANPVT *ppioscanpvt)
```

This routine is called by device or driver support. It is called once for each interrupt source. `scanIoInit` allocates and initializes an array of `io_scan_list` structures; one for each callback priority and puts the address in `pioscanpvt`. Remember that three callback priorities are supported (low, medium, and high). Thus for each interrupt source the structures are illustrated in Figure 17-1:

When `scanAdd` or `scanDelete` are called, they call the device support routine `get_ioint_info` which returns `pioscanpvt`. The `scan_element` is added or deleted from the correct scan list.

17.4.3.2 scanIoRequest

```
scanIoRequest (IOSCANPVT pioscanpvt)
```

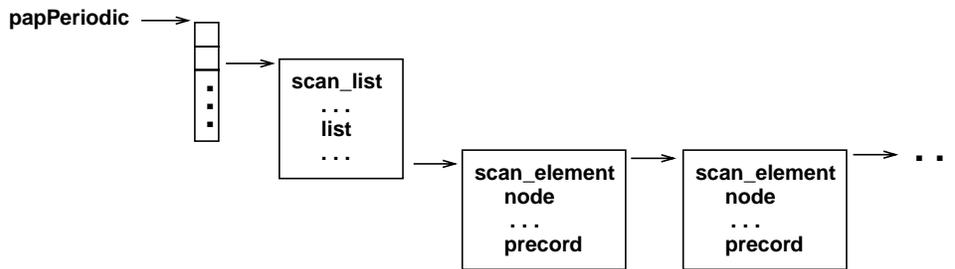


Figure 17-1: Structure after iocInit

This routine is called to request I/O event scanning. It can be called from interrupt level. It looks at each `io_scan_list` referenced by `pioscanpvt` (one for each callback priority) and if any elements are present in the `scan_list` a `callbackRequest` is issued. The appropriate callback task calls routine `ioeventCallback`, which just calls `scanList`.

17.4.4 Periodic Scanning

Periodic scanning is built around the following definitions:

```

static int nPeriodic;
static struct scan_list **papPeriodic;
static int *periodicTaskId;
  
```

`nPeriodic`, which is determined at `iocInit` time, is the number of periodic rates. `papPeriodic` is a pointer to an array of pointers to `scan_lists`. There is an array element for each scan rate. Thus the structure illustrated in Figure 17-1 exists after `iocInit`.

A periodic scan task is created for each scan rate. The following routines implement periodic scanning:

17.4.4.1 initPeriodic

```

initPeriodic()
  
```

This routine first determines the scan rates. It does this by accessing the `SCAN` field of the first record it finds. It issues a call to `dbGetField` with a `DBR_ENUM` request. This returns the menu choices for `SCAN`. From this the periodic rates are determined. The array of pointers referenced by `papPeriodic` is allocated. For each scan rate a `scan_list` is allocated and a `periodicTask` is spawned.

17.4.4.2 periodicTask

```

periodicTask (struct scan_list *psl)
  
```

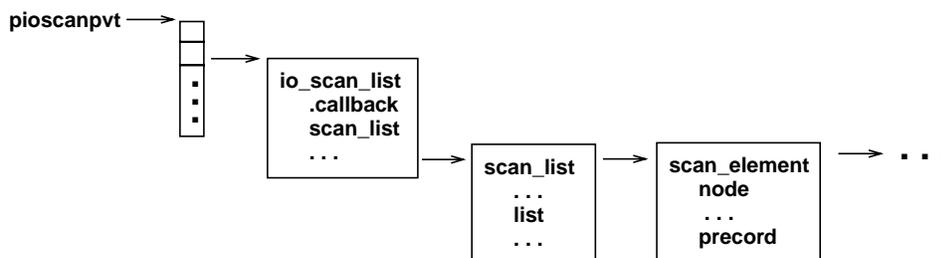


Figure 17-1: Interrupt Source Structure

This task just performs an infinite loop of calling `scanList` and then calling `taskDelay` to wait until the beginning of the next time interval.

17.4.5 Scan Once

17.4.5.1 scanOnce

```
void scanOnce (void *precord)
```

A task `onceTask` waits for requests to issue a `dbProcess` request. The routine `scanOnce` puts the address of the record to be processed in a ring buffer and wakes up `onceTask`.

This routine can be called from interrupt level.

17.4.5.2 SetQueueSize

`scanOnce` places its request on a `vxWorks` ring buffer. This is set by default to 1000 entries. It can be changed by executing the following command in the `vxWorks` startup file.

```
int scanOnceSetQueueSize(int size);
```


Chapter 18: IOC Shell

18.1 Introduction

The EPICS IOC shell is a simple command interpreter which provides a subset of the capabilities of the vxWorks shell. It is used to interpret startup scripts (st.cmd) and to execute commands entered at the console terminal. In most cases vxWorks startup scripts can be interpreted by the IOC shell without modification. The following sections of this chapter describe the operation of the IOC shell from the user's and programmer's points of view.

18.2 IOC Shell Operation

The IOC shell reads lines of input, expands environment variable parameters, breaks the line into commands and arguments then calls functions corresponding to the decoded command. Commands and arguments are separated by one or more 'space' characters. Characters interpreted as spaces include the actual space character and the tab character as well as commas and open and close parentheses. Thus, the command line

```
dbLoadRecords( "db/dbExample1.db" , "user=mrk" )
```

would be interpreted by the IOC shell as the dbLoadRecords command with arguments db/dbExample1.db and user=mrk.

Unrecognized commands result in a diagnostic message but are otherwise ignored. Missing arguments are given a default value (0 for numeric arguments, NULL for string arguments). Extra arguments are ignored.

Unlike the vxWorks shell, string arguments do not have to be enclosed in quotes unless they contain one or more of the 'space' characters in which case one of the quoting mechanisms described in the following section must be used.

18.2.1 Environment variable parameter expansion

Lines of input not beginning with a comment character (#) are searched for character sequences of the form $\${name}$ or $\$(name)$. Such sequences are replaced with the value of the environment variable *name* before any other processing takes place. Expansion is recursive so, for example,

```
epics> epicsEnvSet v1 \${v2}
epics> epicsEnvSet v2 \${v3}
epics> epicsEnvSet v3 somePV
epics> dbpr ${v1}
```

will print information about the somePV process variable (the $\${v1}$ argument to the dbpr command expands to $\${v2}$ which expands to $\${v3}$ which expands to somePV). The backslashes in the definitions are needed to postpone the expansion.

18.2.2 Quoting

Quoting is used to remove the special meaning normally assigned to certain characters and can be used to include `space' or quote characters in arguments. Quoting can not be used to extend a command over more than one input line.

There are three quoting mechanisms: the backslash character, single quotes, and double quotes. A backslash (\) preserves the literal value of the following character. Enclosing characters in single or double quotes preserves the literal value of each character (except a backslash) within the quotes except that parameter expansion still occurs within double quotes. A single quote may occur between double quotes and a double quote may occur between single quotes.

18.2.3 Command-line editing and history

The IOC shell can use the readline or tecla library to obtain input from the console terminal. This provides full command-line editing as well as easy access to previous commands through the command-line history capabilities provided by these libraries. For full details, refer to the readline or tecla library documentation. Command and argument completion is not supported.

If neither the readline nor tecla library is used the only command-line editing and history capabilities will be those supplied by the underlying operating system. The console keyboard driver in Windows, for example, provides its own command-line editing and history commands. On vxWorks the ledLib command-line input libraries are used.

18.2.4 Redirection

The IOC shell recognizes a subset of UNIX shell I/O redirection operators. The redirection operators may precede, or appear anywhere within, or follow a command. Redirections are processed in the order they appear, from left to right. Failure to open or create a file causes the redirection to fail and the command to be ignored.

Redirection of input causes the file whose name results from the expansion of *filename* to be opened for reading on file descriptor *n*, or the standard input (file descriptor 0) if *n* is not specified. The general format for redirecting input is:

```
[n]<filename
```

As a special case, the IOC shell recognizes a standard input redirection appearing by itself (i.e. with no command) as a request to read commands from *filename* until an exit command or EOF is encountered. The IOC shell then resumes reading commands from the current source. Commands read from *filename* are not added to the readline command history. The level of nesting is limited only by the maximum number of files that can be open simultaneously.

Redirection of output causes the file whose name results from the expansion of *filename* to be opened for writing on file descriptor *n*, or the standard output (file descriptor 1) if *n* is not specified. If the file does not exist it is created; if it does exist it is truncated to zero size. The general format for redirecting output is:

```
[n]>filename
```

The general format for appending output is:

```
[n]>>filename
```

Redirection of output in this fashion causes the *filename* to be opened for appending on file descriptor *n*, or the standard output (file descriptor 1) if *n* is not specified. If the file does not exist it is created.

18.2.5 Utility Commands

The IOC shell recognizes the following commands as well as the commands described in chapter 6 (Database Design) and chapter 9 (IOC Test Facilities) among others. In addition, the commands described in the sequencer documentation are recognized.

Command	Description
help [command ...]	Display synopsis of specified commands. Wild-card matching is applied so 'help db*' displays a synopsis of all commands beginning with the letters 'db'. With no arguments display a list of all commands.
#	A '#' in the first column of a line indicates the beginning of a comment which continues to the end of the line
exit	Stop reading commands. When the top-level command interpreter encounters an exit command or end-of-file (EOF) it returns to its caller.
cd <i>directory</i>	Change working directory to <i>directory</i> .
pwd	Print the name of the working directory.
var [<i>name</i> [<i>value</i>]]	If both arguments are present, assign the <i>value</i> to the <i>named</i> variable. If only the <i>name</i> argument is present, print the current value of that variable. If neither argument is present, print the value of all variables registered with the shell. Variables are registered in application database definitions using the <i>variable</i> keyword as described in Section 6.9 on page 102.
show [- <i>level</i>] [<i>task</i> ...]	Show information about specified tasks. If no <i>task</i> arguments are present, show information on all tasks. The <i>level</i> argument controls the amount of information printed. The default <i>level</i> is 0. The <i>task</i> arguments can be task names or task i.d. numbers.
system <i>command_string</i>	Send <i>command_string</i> to the system command interpreter for execution. This command is present only if some application database definition file contains <code>registrar(iocshSystemCommand)</code> and if the system provides a suitable command interpreter (vxWorks does not).
epicsEnvSet <i>name value</i>	Set environment variable <i>name</i> to the specified <i>value</i> .
epicsEnvShow [<i>name</i>]	If no <i>name</i> is specified the names and values of all environment variables will be shown. If a <i>name</i> is specified the value of that environment variable will be shown.
epicsParamShow	Show names and values of all EPICS configuration parameters.
iocLogInit	Initialize IOC logging.
epicsThreadSleep <i>sec</i>	Pause execution of IOC shell for <i>sec</i> seconds.

The *var* command is intended for simple applications such as setting the value of debugging flags. Applications which require more complex expression handling should use the *cexp* package.

A *spy* command to show periodic activity reports is available on RTEMS as part of the RTEMS_UTILS support module. The following changes must be made to add this command to an application.

- Add an RTEMS_UTILS entry to the application configure/RELEASE file.
- Add `spy.dbd` to the list of application dbd files and `rtemsutils` to the list of application libraries in the application Makefile.

18.2.6 ENVIRONMENT VARIABLES

The IOC shell uses the following environment variables to control its operation.

Variable	Description
IOCSH_PS1	Prompt string. Default is "epics>".
IOCSH_HISTSIZ	Number of previous command lines to remember. If the IOCSH_HISTSIZ environment variable is not present the value of the HISTSIZE environment variable is used. In the absence of both environment variables, 10 command lines will be remembered.
TERM, INPUTRC	These and other environment variables are used by the readline and termcap libraries and are described in the documentation for those libraries.

18.3 IOC Shell Programming

The declarations described in this section are included in the <iocsh.h> header file.

18.3.1 Invoking the IOC shell

The prototype for the IOC shell command interpreter is:

```
int iocsh (const char *pathname);
```

The argument is the name of the file from which commands are to be read. If the *pathname* argument is NULL, commands are read from the standard input and prompts are issued to the standard output. Commands are read until an exit command is encountered or until end-of-file is reached at which point iocsh returns a value of 0. If the specified file can not be opened iocsh returns -1.

The IOC shell can be invoked from the vxWorks shell, either from within a vxWorks startup script or from vxWorks command-line interpreter, as

```
iocsh "filename"
```

to read from an IOC shell script. It can also be invoked from the vxWorks command-line interpreter with no argument in which case the IOC shell takes over the duties of command-line interaction.

To execute a single IOC shell command from a vxWorks startup script or command line the

```
iocshCmd "iocsh command string"
```

command can be used.

18.3.2 Registering Commands

Commands must be registered before they can be recognized by the IOC shell. Registration is achieved by calling the registration function:

```
void iocshRegister(const iocshFuncDef *piocshFuncDef, iocshCallFunc func);
```

The first argument is a pointer to a data structure which describes the command and any arguments it may take. The second argument is a pointer to a function which will be called by iocsh when the corresponding command is encountered.

The command is described by the `iocshFuncDef` structure:

```
struct iocshFuncDef {
    const char *name;
    int nargs;
    const iocshArg * const *arg;
};
```

The `name` element is the name of the command. The `arg` element is a pointer to an array of pointers to structures each of which defines a single argument. The `nargs` element declares the number of entries in the array of pointers to the argument descriptions. If `nargs` is zero, `arg` can be `NULL`. The structures which define each of the arguments is:

```
struct iocshArg {
    const char *name;
    iocshArgType type;
}iocshArg;
```

The `name` element is used by the `help` command to print a synopsis for the command. The `type` element describes the type of the argument and takes one of the following values:

Type Specifier	Description
<code>iocshArgInt</code>	The argument will be converted to an integer value.
<code>iocshArgDouble</code>	The argument will be converted to a double-precision floating point value.
<code>iocshArgString</code>	The argument will be left as a string. The memory used to hold the string is 'owned' by <code>iocsh</code> and will be reused once the handler function returns.
<code>iocshArgPersistentString</code>	A copy of the argument will be made and a pointer to the copy will be passed to the handler. The called function can release this copy by using the pointer as an argument to <code>free()</code> .
<code>iocshArgPdbbase</code>	The argument must be <code>pdbbase</code> .
<code>iocshArgArgv</code>	An arbitrary number of arguments is expected. Subsequent <code>iocshArg</code> structures will be ignored.

The 'handler' function which is called when its corresponding command is recognized should be of the form:

```
void showCallFunc(const iocshArgBuf *args);
```

The argument to the handler function is a pointer to an array of unions. The number of elements in this array is equal to the number of arguments specified in the structure describing the command. The type and name of the union element which contains the argument value depends on the 'type' element of the corresponding argument descriptor:

Type Specifier	Type	Union element
<code>iocshArgInt</code>	<code>int</code>	<code>args[i].ival</code>
<code>iocshArgDouble</code>	<code>double</code>	<code>args[i].dval</code>
<code>iocshArgString</code> <code>iocshArgPersistentString</code>	<code>char *</code>	<code>args[i].sval</code>
<code>iocshArgPdbbase</code>	<code>void *</code>	<code>args[i].vval</code>

Type Specifier	Type	Union element
iocshArgArgv	int char **	args[i].aval.ac args[i].aval.av

If an `iocshArgArgv` argument type is present it is often the first and only argument specified for the command. In this case, `args[0].aval.av[0]` will be the name of the command, `args[0].aval.av[1]` will be the first argument, and so on.

18.3.3 Automatic Command Registration

A C++ static constructor can be used to register IOC shell commands before the EPICS application begins. The following example shows how the ‘`epicsThreadSleep`’ command could be described and registered.

```
#include <iocsh.h>

static const iocshArg epicsThreadSleepArg0 = { "seconds",iocshArgDouble};
static const iocshArg *const epicsThreadSleepArgs[1] = {&epicsThreadSleepArg0};
static const iocshFuncDef epicsThreadSleepFuncDef =
    {"epicsThreadSleep",1,epicsThreadSleepArgs};
static void epicsThreadSleepCallFunc(const iocshArgBuf *args)
{
    epicsThreadSleep(args[0].dval);
}

class IocshRegister {
public:
    IocshRegister() {
        iocshRegister(epicsThreadSleepFuncDef, epicsThreadSleepCallFunc);
    }
};

static IocshRegister iocshRegisterObj;
```

18.3.4 Registrar Command Registration

As of EPICS release R3.14.2, it is recommended that a registrar function be used to register iocsh commands. An application database description file uses the **registrar** keyword to specify a function which will be called from the EPICS initialization code during the application startup process. This function then calls `iocshRegister` to register commands with the iocsh.

The following code fragments shows how this can be performed for an example driver.

```
#include <iocsh.h>
#include <epicsExport.h>

.
.
<< Xxx Driver Code >>
.
.

static void
drvXxxRegistrar(void)
```

```
{  
    iocshRegister(&drvXxxConfigureFuncDef, drvXxxConfigureCallFunc);  
    .  
    .  
}  
epicsExportRegistrar(drvXxxRegistrar);
```

To include this driver in an application a developer would then add
`registrar(drvXxxRegistrar)`
to an application database description file.

Chapter 19: *libCom*

This chapter and the next describe the facilities provided in `<base>/src/libCom`. This chapter describes facilities which are platform independent. The next chapter describes facilities which have different implementations on different platforms.

19.1 *bucketLib*

`bucketLib.h` describes a hash facility for integers, pointers, and strings. It is used by the Channel Access Server. It is currently undocumented.

19.2 *calc*

`postfix.h` defines several macros and the routines used by the calculation record type `calcRecord`, access security, and other code, to compile and evaluate mathematical expressions. The syntax of the infix expressions accepted is described in Section 19.2.1 on page 254 below.

```
long postfix(const char *psrc, char *ppostfix, short *perror);
long calcArgUsage(const char *ppostfix, unsigned long *pinputs,
                 unsigned long *pstores);
const char * calcErrorStr(short error);
long calcPerform(double *parg, double *presult, const char *ppostfix);
```

The `postfix()` routine converts an expression from infix to postfix notation. It is the callers's responsibility to make sure that `ppostfix` points to sufficient storage to hold the postfix expression; the macro `INFIX_TO_POSTFIX_SIZE(n)` can be used to calculate an appropriate buffer from the length of the infix string. There is no longer a maximum length to the input expression that can be accepted, although there are internal limits to the complexity of the expressions that can be converted and evaluated. If `postfix()` returns a non-zero value it will have placed an error code at the location pointed to by `perror`. The error codes used are defined in `postfix.h` as a series of macros with names starting `CALC_ERR_`, but a string representation of the error code is more useful and can be obtained by passing the value to the `calcErrorStr()` routine, which returns a static error message string explaining the error.

Software using the `calc` subsystem may need to know what expression arguments are used and/or modified by a particular expression. It can now discover this from the postfix string by calling `calcArgUsage()`, which takes two pointers `pinputs` and `pstores` to a pair of unsigned long bitmaps which return that information to the caller (passing a NULL value for either of these pointers is legal). The least significant bit (bit 0) of the bitmap at `*pinputs` will be set if the expression depends on the argument A, and so on through bit 11 for the argument L. Similarly, bit 0 of the bitmap at `*pstores` will be set if the expression assigns a value to the argument A. An argument that is not used until after a value has been assigned to it will not be set in the `pinputs` bitmap, thus the bits can be used to determine whether a value needs to be supplied for their associated argument or not (for the purposes of evaluating the expression at least; other considerations make it desirable to still fetch all input values in the case of the `CALC` record). The return value from `calcArgUsage()` will be non-zero if the `ppostfix` expression was illegal, otherwise 0.

The postfix expression is evaluated by calling the `calcPerform()` routine, which returns the status values 0 for OK, or non-zero if an error is discovered during the evaluation process.

The arguments to `calcPerform()` are:

- parg* - Pointer to an array of double values for the arguments A-L that can appear in the expression. Note that the argument values may be modified if the expression uses the assignment operator.
- presult* - Where to put the calculated result, which may be a NaN or Infinity.
- ppostfix* - The postfix expression created by `postfix()`.

19.2.1 Infix Expression Syntax

The infix expressions that can be used are very similar to the C expression syntax, but with some additions and subtle differences in operator meaning and precedence. The string may contain a series of expressions separated by a semi-colon character `' ; '` any one of which may actually provide the calculation result; however all of the other expressions included must assign their result to a variable. All alphabetic elements described below are case independent, so upper and lower case letters may be used and mixed in the variable and function names as desired. Spaces may be used anywhere within an expression except between the characters that make up a single expression element.

19.2.1.1 Numeric Literals

The simplest expression element is a numeric literal, any (positive) number expressed using the standard floating point syntax that can be stored as a double precision value. This now includes the values `Infinity` and `NaN` (not a number). Note that negative numbers are actually encoded as a positive literal to which the unary negate operator is applied.

Examples:

```
1
2.718281828459
Inf
```

19.2.1.2 Constants

There are three trigonometric constants available to any expression which return a value:

- `pi` returns the value of the mathematical constant π .
- `D2R` evaluates to $\pi/180$ which, when used as a multiplier, converts an angle from degrees to radians.
- `R2D` evaluates to $180/\pi$ which as a multiplier converts an angle from radians to degrees.

19.2.1.3 Variables

Variables are used to provide inputs to an expression, and are named using the single letters A through L inclusive. The software that makes use of the expression evaluation code should document how the individual variables are given values; for the calc record type the input links INPA through INPL can be used to obtain these from other record fields.

19.2.1.4 Variable Assignment Operator

Recently added is the ability to assign the results of an expression to a variable, which can be used later in another expression. The variable assignment operator is the character pair `:=` and must immediately follow the name of the variable to receive the expression value. Since the infix string must return exactly one value, every expression string must have exactly one expression that is not an assignment, which can appear anywhere in the string. Expressions within the string are separated by semi-colon characters.

Examples:

```
B; B:=A
i:=i+1; a*sin(i*D2R)
```

19.2.1.5 Arithmetic Operators

The usual binary arithmetic operators are provided: + - * and / with their usual relative precedence and left-to-right associativity, and - may also be used as a unary negate operator where it has a higher precedence and associates from right to left. There is no unary plus operator, so numeric literals cannot begin with a + sign.

Examples:

```
a*b + c
a/-4 - b
```

Three other binary operators are also provided: % is the integer modulo operator, and the synonymous operators ** and ^ raise their left operand to the power of the right operand. % has the same precedence and associativity as * and /, while the power operators associate left-to-right and have a precedence in between * and unary minus.

Examples:

```
e:=a%10; d:=a/10%10; c:=a/100%10; b:=a/1000%10; b*4096+c*256+d*16+e
sqrt(a**2 + b**2)
```

19.2.1.6 Algebraic Functions

Various algebraic functions are available which take one or two parameters inside parentheses. The parameter separator is a comma. *While it is currently possible to omit the parentheses for single parameter functions in some expressions, this usage is deprecated and may be removed from future versions of the expression parser.*

- Absolute value: `abs(a)`
- Exponential e^a : `exp(a)`
- Logarithm, base 10: `log(a)`
- Natural logarithm (base e): `ln(a)` or `loge(a)`
- Maximum value: `max(a, b)`
- Minimum value: `min(a, b)`
- Square root: `sqr(a)` or `sqrt(a)`

19.2.1.7 Trigonometric Functions

Standard circular trigonometric functions, with angles expressed in radians:

- Sine: `sin(a)`
- Cosine: `cos(a)`
- Tangent: `tan(a)`
- Arcsine: `asin(a)`
- Arccosine: `acos(a)`
- Arctangent: `atan(a)`
- 2 parameter arctangent: `atan2(a, b)` — *Note that these arguments are the reverse of the ANSI C function, such that while C would return `arctan(a/b)` the calc expression engine returns `arctan(b/a)`*

19.2.1.8 Hyperbolic Trigonometry

The basic hyperbolic functions are provided, but no inverse functions (which are not provided by the ANSI C math library either).

- Hyperbolic sine: `sinh(a)`
- Hyperbolic cosine: `cosh(a)`
- Hyperbolic tangent: `tanh(a)`

19.2.1.9 Numeric Functions

The numeric functions perform operations related to the floating point numeric representation and truncation or rounding.

- Round up to next integer: `ceil(a)`
- Round down to next integer: `floor(a)`
- Round to nearest integer: `nint(a)`
- Test for infinite result: `isinf(a)`
- Test for non-numeric result: `isnan(a)`
- Test for finite, numeric result: `finite(a)`
- Random number between 0 and 1: `rndm`

19.2.1.10 Boolean Operators

These operators regard their arguments as true or false, where 0.0 is false and any other value is true.

- Boolean and: `a && b`
- Boolean or: `a || b`
- Boolean not: `!a`

19.2.1.11 Bitwise Operators

The bitwise operators convert their arguments to an integer (by truncation), perform the appropriate bitwise operation and convert back to a floating point value. Note that unlike in C, `^` is *not* a bitwise exclusive-or operator.

- Bitwise and: `a & b` or `a and b`
- Bitwise or: `a | b` or `a or b`
- Bitwise exclusive or: `a xor b`
- Bitwise not (ones complement): `~a` or `not a`
- Bitwise left shift: `a << b`
- Bitwise right shift: `a >> b`

19.2.1.12 Relational Operators

Standard numeric comparisons between two values:

- Less than: `a < b`
- Less than or equal to: `a <= b`
- Equal to: `a = b` or `a == b`
- Greater than or equal to: `a >= b`
- Greater than: `a > b`
- Not equal to: `a != b` or `a # b`

19.2.1.13 Conditional Operator

Expressions can use the C conditional operator, which has a lower precedence than all of the other operators except for the assignment operator.

- *condition ? true result : false result*

Example:

```
a < 360 ? a+1 : 0
```

19.2.1.14 Parentheses

Sub-expressions can be placed within parentheses to override operator precedence rules. Parentheses can be nested to any depth, but the intermediate value stack used by the expression evaluation engine is limited to 80 results (which would require an expression at least 321 characters long to reach).

19.3 *cppStd*

This subdirectory of libCom is intended for facilities such as class and function templates that implement parts of the ISO standard C++ library where such facilities are not available or not efficient on all the target platforms on which EPICS is supported. EPICS does not make use of the C++ container templates because the large number of memory allocation and deletion operations that these use causes memory pool fragmentation on some platforms, threatening the lifetime of an individual IOC.

19.4 *epicsExit*

```
void epicsExit(int status);
void epicsExitCallAtExits(void);
void epicsAtExit(void (*epicsExitFunc)(void *arg), void *arg);
void epicsExitCallAtThreadExits(void);
int epicsAtThreadExit(void (*epicsExitFunc)(void *arg), void *arg);
```

This is an extended replacement for the Posix exit and atexit facility, which also provides an argument to pass to the exit handlers. This facility was created because of problems on vxWorks and windows with the implementation of atexit, i.e. neither of these systems implements exit and atexit according to the POSIX standard.

Method	Meaning
epicsExit	This calls epicsExitCallAtExits and then passes status on to exit.
epicsExitCallAtExits	This calls each of the functions registered by prior calls to epicsAtExit, in reverse order of their registration. Most applications will not call this routine directly.
epicsAtExit	Register a function and an associated context parameter, to be called with the given parameter when epicsExitCallAtExits is invoked.
epicsExitCallAtThreadExits	This calls each of the functions that were registered by the current thread calling epicsAtThreadExit, in reverse order of the function registration. This routine is called automatically when an epicsThread's main entry method returns, but will not be run if the thread is stopped by other means.
epicsAtThreadExit	Register a function and an associated context parameter. The function will be called with the given parameter when epicsExitCallAtThreadExits is invoked by the current thread.

19.4.1 epicsAlgorithm

epicsAlgorithm.h contains a few templates that are also available in the C++ standard header algorithm, but are provided here in a much smaller file – algorithm contains many templates for sorting and searching. If all you need from there is std::min(), std::max() and/or std::swap() your code will compile faster if you include epicsAlgorithm.h and use epicsMin(), epicsMax() and epicsSwap() instead.

template <class T>	Meaning
const T& epicsMin(const T& a, const T& b)	Returns the smaller of <i>a</i> or <i>b</i> compared using <i>a</i> < <i>b</i> . Handles NaNs correctly.
const T& epicsMax(const T& a, const T& b)	Returns the larger of <i>a</i> or <i>b</i> compared using <i>a</i> < <i>b</i> . Handles NaNa correctly.
void epicsSwap(T& a, T& b)	Swaps the values of <i>a</i> and <i>b</i> ; T must have a copy-constructor and operator=.

19.5 cvtFast

cvtFast.h provides routines for converting various numeric types into an ascii string. They offer a combination of speed and convenience not available with sprintf().

```
/* These functions return the number of ASCII characters generated */
int cvtFloatToString(float value, char *pstr, unsigned short precision);
int cvtDoubleToString(double value, char *pstr, unsigned short prec);
int cvtFloatToExpString(float value, char *pstr, unsigned short prec);
int cvtDoubleToExpString(double value, char *pstr, unsigned short prec);
int cvtFloatToCompactString(float value, char *pstr, unsigned short prec);
int cvtDoubleToCompactString(double value, char *pstr, unsigned short prec);
int cvtCharToString(char value, char *pstring);
int cvtUcharToString(unsigned char value, char *pstr);
int cvtShortToString(short value, char *pstr);
int cvtUshortToString(unsigned short value, char *pstr);
int cvtLongToString(epicsInt32 value, char *pstr);
int cvtUlongToString(epicsUInt32 value, char *pstr);
int cvtLongToHexString(epicsInt32 value, char *pstr);
int cvtLongToOctalString(epicsInt32 value, char *pstr);
unsigned long cvtBitsToUlong(
    epicsUInt32 src,
    unsigned bitFieldOffset,
    unsigned bitFieldLength);
unsigned long cvtUlongToBits(
    epicsUInt32 src,
    epicsUInt32 dest,
    unsigned bitFieldOffset,
    unsigned bitFieldLength);
```

19.6 cxxTemplates

This directory contains the following C++ template headers:

- `resourceLib.h` - A C++ hash facility that implements the same functionality as `bucketLib`
- `tsBTree.h` - Binary tree.
- `tsDLLList.h` - Double Linked List
- `tsFreeList.h` - Free List for efficient new/delete
- `tsMinMax.h` - min and max.
- `tsSLLList.h` - Single Linked List

Currently these are only being used by Channel Access Clients and the portable Channel Access Server. It has not been decided if any of these will remain in libCom.

19.7 dbmf

`dbmf.h` (Database Macro/Free) describes a facility that prevents memory fragmentation when memory is allocated and then freed a short time later.

Routines within `iocCore` like `dbLoadDatabase()` have the following attributes:

- They issue many calls to `malloc()` followed a short time later by a call to `free()`.
- Between a call to `malloc()` and the associated `free()`, an additional call to `malloc()` is made that does NOT have an associated `free()`.

In some environments, e.g. `vxWorks`, this behavior causes severe memory fragmentation.

The `dbmf` facility stops the memory fragmentation. It should NOT be used by code that allocates storage and then keeps it for a considerable period of time before releasing. Such code can use the `freeList` library described below.

```
int dbmfInit(size_t size, int chunkItems);
void *dbmfMalloc(size_t bytes);
void dbmfFree(void* bytes);
void dbmfFreeChunks(void);
int dbmfShow(int level);
```

Routine	Meaning
<code>dbmfInit()</code>	Initialize the facility. Each time <code>malloc()</code> must be called <code>size*chunkItems</code> bytes are allocated. <code>size</code> is the maximum size request from <code>dbmfMalloc()</code> that will be allocated from the <code>dbmf</code> pool. If <code>dbmfInit()</code> was not called before one of the other routines then it is automatically called with <code>size=64</code> and <code>chunkItems=10</code> .
<code>dbmfMalloc()</code>	Allocate memory. If <code>bytes</code> is $>$ <code>size</code> then <code>malloc()</code> is used to allocate the memory.
<code>dbmfFree()</code>	Free the memory allocated by <code>dbmfMalloc()</code> .
<code>dbmfFreeChunks()</code>	Free all chunks that have contain only free items.
<code>dbmfShow()</code>	Show the status of the <code>dbmf</code> memory pool.

19.8 *ellLib*

ellLib.h describes a double linked list library. It provides functionality similar to the vxWorks lstLib library. See the vxWorks documentation for details. There is an ellXXX() routine to replace most vxWorks lstXXX() routines.

```
typedef struct ELLNODE {
    struct ELLNODE *next;
    struct ELLNODE *previous;
} ELLNODE;

typedef struct ELLLIST {
    ELLNODE node;
    int count;
} ELLLIST;

void ellInit (ELLLIST *pList);
int ellCount (ELLLIST *pList);
ELLNODE *ellFirst (ELLLIST *pList);
ELLNODE *ellLast (ELLLIST *pList);
ELLNODE *ellNext (ELLNODE *pNode);
ELLNODE *ellPrevious (ELLNODE *pNode);
void ellAdd (ELLLIST *pList, ELLNODE *pNode);
void ellConcat (ELLLIST *pDstList, ELLLIST *pAddList);
void ellDelete (ELLLIST *pList, ELLNODE *pNode);
void ellExtract (ELLLIST *pSrcList, ELLNODE *pStartNode,
                ELLNODE *pEndNode, ELLLIST *pDstList);
ELLNODE *ellGet (ELLLIST *pList);
void ellInsert (ELLLIST *pList, ELLNODE *pPrev, ELLNODE *pNode);
ELLNODE *ellNth (ELLLIST *pList, int nodeNum);
ELLNODE *ellNStep (ELLNODE *pNode, int nStep);
int ellFind (ELLLIST *pList, ELLNODE *pNode);
void ellFree (ELLLIST *pList);
void ellVerify (ELLLIST *pList);
```

19.9 *epicsRingBytes*

epicsRingBytes.h contains

```
epicsRingBytesId epicsRingBytesCreate(int nbytes);
void epicsRingBytesDelete(epicsRingBytesId id);
int epicsRingBytesGet(epicsRingBytesId id, char *value,int nbytes);
int epicsRingBytesPut(epicsRingBytesId id, char *value,int nbytes);
void epicsRingBytesFlush(epicsRingBytesId id);
int epicsRingBytesFreeBytes(epicsRingBytesId id);
int epicsRingBytesUsedBytes(epicsRingBytesId id);
int epicsRingBytesSize(epicsRingBytesId id);
int epicsRingBytesIsEmpty(epicsRingBytesId id);
int epicsRingBytesIsFull(epicsRingBytesId id);
```

Method	Meaning
epicsRingBytesCreate()	Create a new ring buffer of size <i>nbytes</i> . The returned epicsRingBytesId is passed to the other ring methods.
epicsRingBytesDelete()	Delete the ring buffer and free any associated memory.
epicsRingBytesGet()	Move up to <i>nbytes</i> from the ring buffer to <i>value</i> . The number of bytes actually moved is returned.
epicsRingBytesPut()	Move up to <i>nbytes</i> from <i>value</i> to the ring buffer. The number of bytes actually moved is returned.
epicsRingBytesFlush()	Make the ring buffer empty.
epicsRingBytesFreeBytes()	Return the number of free bytes in the ring buffer.
epicsRingBytesUsedBytes()	Return the number of bytes currently stored in the ring buffer.
epicsRingBytesSize()	Return the size of the ring buffer, i.e., <i>nbytes</i> specified in the call to epicsRingBytesCreate().
epicsRingBytesIsEmpty()	Return (true, false) if the ring buffer is currently empty.
epicsRingBytesIsFull()	Return (true, false) if the ring buffer is currently empty.

epicsRingBytes has the following properties:

- For a ring buffer with a single writer it is not necessary to lock epicsRingBytesPut() calls.
- For a ring buffer with a single reader it is not necessary to lock epicsRingBytesGet() calls.
- epicsRingBytesFlush() should only be used when both gets and puts are locked out.
- Readers and writers must be prepared to deal with transfers smaller than the requested size.

19.10 epicsRingPointer

epicsRingPointer.h describes a C++ and a C facility for a commonly used type of ring buffer.

19.10.1 C++ Interface

EpicsRingPointer provides methods for creating and using ring buffers (first in first out circular buffers) that store pointers. It is designed so that a writer thread and reader thread can access the ring simultaneously without requiring mutual exclusion.

```
template <class T>
class epicsRingPointer {
public:
    epicsRingPointer(int size);
    ~epicsRingPointer();
    bool push(T *p);
    T* pop();
    void flush();
    int getFree() const;
    int getUsed() const;
    int getSize() const;
```

```

    bool isEmpty() const;
    bool isFull() const;

private: // Prevent compiler-generated member functions
    // default constructor, copy constructor, assignment operator
    epicsRingPointer();
    epicsRingPointer(const epicsRingPointer &);
    epicsRingPointer& operator=(const epicsRingPointer &);

private: // Data
    ...
};

```

An epicsRingPointer cannot be assigned to, copy-constructed, or constructed without giving the *size* argument. The C++ compiler will object to some of the statements below:

```

epicsRingPointer rp0(); // Error: default constructor is private
epicsRingPointer rp1(10); // OK
epicsRingPointer rp2(t1); // Error: copy constructor is private
epicsRingPointer *prp; // OK, pointer
*prp = rp1; // Error: assignment operator is private
prp = &rp1; // OK, pointer assignment and address-of

```

Method	Meaning
epicsRingPointer()	Constructor. The size is the maximum number of elements (pointers) that can be stored in the ring.
~epicsRingPointer()	Destructor.
push()	Push a new entry on the ring. It returns (false,true) is (failure, success). Failure means the ring was full. If a single writer is present it does not have to use a lock while performing the push. If multiple writers are present they must use a common lock while issuing the push.
pop()	Take a element off the ring. It returns 0 (null) if the ring was empty. If a single reader is present it does not have to lock while issuing the pop. If multiple readers are present they must use a common lock while issuing the pop.
flush()	Remove all elements from the ring. If this operation is performed then all access to the ring should be locked.
getFree()	Return the amount of empty space in the ring, i.e. how many additional elements it can hold.
getUsed()	Return the number of elements stored on the ring
getSize()	Return the size of the ring, i.e. the value of <i>size</i> specified when the ring was created.
isEmpty()	Returns true if the ring is empty, else false.
isFull()	Returns true if the ring is full, else false.

19.10.2 C interface

```

typedef void *epicsRingPointerId;
epicsRingPointerId epicsRingPointerCreate(int size);

```

```

void epicsRingPointerDelete(epicsRingPointerId id);
/*epicsRingPointerPop returns 0 if the ring was empty */
void * epicsRingPointerPop(epicsRingPointerId id) ;
/*epicsRingPointerPush returns (0,1) if p (was not, was) put on ring*/
int epicsRingPointerPush(epicsRingPointerId id,void *p);
void epicsRingPointerFlush(epicsRingPointerId id);
int epicsRingPointerGetFree(epicsRingPointerId id);
int epicsRingPointerGetUsed(epicsRingPointerId id);
int epicsRingPointerGetSize(epicsRingPointerId id);
int epicsRingPointerIsEmpty(epicsRingPointerId id);
int epicsRingPointerIsFull(epicsRingPointerId id);

```

Each C function corresponds to one of the C++ methods.

19.11 *epicsTimer*

`epicsTimer.h` describes a C++ and a C timer facility.

19.11.1 C++ Interface

19.11.1.1 `epicsTimerNotify` and `epicsTimer`

```

class epicsTimerNotify {
public:
    enum restart_t { noRestart, restart };
    class expireStatus {
    public:
        expireStatus ( restart_t );
        expireStatus ( restart_t, const double &expireDelaySec );
        bool restart () const;
        double expirationDelay () const;
    private:
        double delay;
    };
    virtual ~epicsTimerNotify ();
    // return noRestart OR return expireStatus ( restart, 30.0 /* sec */ );
    virtual expireStatus expire ( const epicsTime & currentTime ) = 0;
    virtual void show ( unsigned int level ) const;
};

class epicsTimer {
public:
    virtual void destroy () = 0; // requires existence of timer queue
    virtual void start ( epicsTimerNotify &, const epicsTime & ) = 0;
    virtual void start ( epicsTimerNotify &, double delaySeconds ) = 0;
    virtual void cancel () = 0;
    struct expireInfo {
        expireInfo ( bool active, const epicsTime & expireTime );
        bool active;
        epicsTime expireTime;
    };
};

```

```
};
virtual expireInfo getExpireInfo () const = 0;
double getExpireDelay ();
virtual void show ( unsigned int level ) const = 0;
protected:
virtual ~epicsTimer () = 0; // use destroy
};
```

Method	Meaning
epicsTimerNotify expire()	Code using an epicsTimer must include a class that inherits from epicsTimerNotify. The derived class must implement the method expire(), which is called by the epicsTimer when the associated timer expires. epicsTimerNotify defines a class expireStatus which makes it easy to implement both one shot and periodic timers. A one-shot expire() returns with the statement: <pre>return(noRestart);</pre> A periodic timer returns with a statement like: <pre>return(restart, 10.0);</pre> where is second argument is the delay until the next callback.
epicsTimer	epicsTimer is an abstract base class. An epics timer can only be created by calling createTimer, which is a method of epicsTimerQueue.
destroy	This is provided instead of a destructor. This will automatically call cancel before freeing all resources used by the timer.
start()	Starts the timer to expire either at the specified time or the specified number of seconds in the future. If the timer is already active when start is called, it is first canceled.
cancel()	If the timer is scheduled, cancel it. If it is not scheduled do nothing. Note that if the expire() method is already running, this call delays until the expire() completes.
getExpireInfo	Get expireInfo, which says if timer is active and if so when it expires.
getExpireDelay()	Return the number of seconds until the timer will expire. If the timer is not active it returns DBL_MAX
show()	Display info about object.

19.11.1.2 epicsTimerQueue

```
class epicsTimerQueue {
public:
virtual epicsTimer & createTimer () = 0;
virtual void show ( unsigned int level ) const = 0;
protected:
virtual ~epicsTimerQueue () = 0;
};
```

Method	Meaning
createTimer()	This is a "factory" method to create timers which use this queue.
show()	Display info about object

19.11.1.3 epicsTimerQueueActive

```

class epicsTimerQueueActive : public epicsTimerQueue {
public:
    static epicsTimerQueueActive & allocate (
        bool okToShare, unsigned threadPriority = epicsThreadPriorityMin + 10 );
    virtual void release () = 0;
protected:
    virtual ~epicsTimerQueueActive () = 0;
};

```

Method	Meaning
allocate()	This is a "factory" method to create a timer queue. If okToShare is (true,false) then a (shared, separate) thread will manage the timer requests. If the okToShare constructor parameter is true then if a timer queue is already running at the specified priority then it will be referenced for shared use by the application, and an independent timer queue will not be created. Editorial note: It is useful for two independent timer queues to run at the same priority if there are multiple processors, or if there is an application with well behaved timer expire functions that needs to be independent of applications with computationally intensive, mutex locking, or IO blocking timer expire functions.
release()	Release the queue, i.e. the calling facility will no longer use the queue. The caller MUST ensure that it does not own any active timers. When the last facility using the queue calls release, all resources used by the queue are freed.

19.11.1.4 epicsTimerQueueNotify and epicsTimerQueuePassive

These two classes manage a timer queue for single threaded applications. Since it is single threaded, the application is responsible for requesting that the queue be processed.

```

class epicsTimerQueueNotify {
public:
    // called when a new timer is inserted into the queue and the
    // delay to the next expire has changed
    virtual void reschedule () = 0;
    // if there is a quantum in the scheduling of timer intervals
    // return this quantum in seconds. If unknown then return zero.
    virtual double quantum () = 0;
protected:
    virtual ~epicsTimerQueueNotify () = 0;
};

class epicsTimerQueuePassive {
public:
    static epicsTimerQueuePassive & create ( epicsTimerQueueNotify & );
    virtual ~epicsTimerQueuePassive () = 0;
    // process returns the delay to the next expire
    virtual double process (const epicsTime & currentTime) = 0;
};

```

Method	Meaning
epicsTimerQueueNotify reschedule()	The virtual function epicsTimerQueueNotify::reschedule() is called when the delay to the next timer to expire on the timer queue changes.
epicsTimerQueueNotify quantum	The virtual function epicsTimerQueueNotify::quantum() returns the timer expire interval scheduling quantum in seconds. This allows different types of timer queues to use application specific timer expire delay scheduling policies. The implementation of epicsTimerQueueActive employs epicsThreadSleep() for this purpose, and therefore epicsTimerQueueActive::quantum() returns the returned value from epicsThreadSleepQuantum(). Other types of timer queues might choose to schedule timer expiration using specialized hardware interrupts. In this case epicsTimerQueueNotify::quantum() might return a value reflecting the precision of a hardware timer. If unknown, then epicsTimerQueueNotify::quantum() should return zero.
epicsTimerQueuePassive	epicsTimerQueuePassive is an abstract base class so cannot be instantiated directly, but contains a static member function to create a concrete passive timer queue object of a (hidden) derived class.
create()	A "factory" method to create a non-threaded timer queue. The calling software also passes an object derived from epicsTimerQueueNotify to receive reschedule() callbacks.
~epicsTimerQueuePassive()	Destructor. The caller MUST ensure that it does not own any active timers, i.e. it must cancel any active timers before deleting the epicsTimerQueuePassive object.
process()	This calls expire() for all timers that have expired. The facility that creates the queue MUST call this. It returns the delay until the next timer will expire.

19.11.2 C Interface

```

typedef struct epicsTimerForC * epicsTimerId;
typedef void ( *epicsTimerCallback ) ( void *pPrivate );

/* thread managed timer queue */
typedef struct epicsTimerQueueActiveForC * epicsTimerQueueId;
epicsTimerQueueId epicsTimerQueueAllocate(
    int okToShare, unsigned int threadPriority );
void epicsTimerQueueRelease ( epicsTimerQueueId );
epicsTimerId epicsTimerQueueCreateTimer ( epicsTimerQueueId queueid,
    epicsTimerCallback callback, void *arg );
void epicsTimerQueueDestroyTimer( epicsTimerQueueId queueid, epicsTimerId id );
void epicsTimerQueueShow ( epicsTimerQueueId id, unsigned int level );

/* passive timer queue */
typedef struct epicsTimerQueuePassiveForC * epicsTimerQueuePassiveId;
typedef void ( *epicsTimerQueueNotifyReschedule ) ( void *pPrivate );
typedef double ( * epicsTimerQueueNotifyQuantum ) ( void * pPrivate );
epicsTimerQueuePassiveId epicsTimerQueuePassiveCreate(
    epicsTimerQueueNotifyReschedule, epicsTimerQueueNotifyQuantum,
    void *pPrivate );
void epicsTimerQueuePassiveDestroy ( epicsTimerQueuePassiveId );
epicsTimerId epicsTimerQueuePassiveCreateTimer (epicsTimerQueuePassiveId queueid,
    epicsTimerCallback pCallback, void *pArg );
void epicsTimerQueuePassiveDestroyTimer (
    epicsTimerQueuePassiveId queueid, epicsTimerId id );

```

```
double epicsTimerQueuePassiveProcess ( epicsTimerQueuePassiveId );
void epicsTimerQueuePassiveShow(epicsTimerQueuePassiveId id,unsigned int level);
/* timer */
void epicsTimerStartTime(epicsTimerId id, const epicsTimeStamp *pTime);
void epicsTimerStartDelay(epicsTimerId id, double delaySeconds);
void epicsTimerCancel ( epicsTimerId id );
double epicsTimerGetExpireDelay ( epicsTimerId id );
void epicsTimerShow ( epicsTimerId id, unsigned int level );
```

The C interface provides most of the facilities as the C++ interface. It does not support the periodic timer features. The typedefs `epicsTimerQueueNotifyReschedule` and `epicsTimerQueueNotifyQuantum` are the "C" interface equivalents to `epicsTimerQueueNotify::reschedule()` and `epicsTimerQueueNotify::quantum()`.

19.11.3 Example

This example allocates a timer queue and two objects which have a timer that uses the queue. Each object is requested to schedule itself. The `expire()` callback just prints the name of the object. After scheduling each object the main thread just sleeps long enough for each expire to occur and then just returns after releasing the queue.

```
#include <stdio.h>
#include "epicsTimer.h"

class something : public epicsTimerNotify {
public:
    something(const char* nm,epicsTimerQueueActive &queue)
        : name(nm), timer(queue.createTimer()) {}
    virtual ~something() { timer.destroy();}
    void start(double delay) {timer.start(*this,delay);}
    virtual expireStatus expire(const epicsTime & currentTime) {
        printf("%s\n",name);
        currentTime.show(1);
        return(noRestart);
    }
private:
    const char* name;
    epicsTimer &timer;
};

void epicsTimerExample()
{
    epicsTimerQueueActive &queue = epicsTimerQueueActive::allocate(true);
    {
        something first("first",queue);
        something second("second",queue);

        first.start(1.0);
        second.start(1.5);
        epicsThreadSleep(2.0);
    }
    queue.release();
}
```

19.11.4 C Example

This example shows how C programs can use EPICS timers.

```
#include <stdio.h>
#include <epicsTimer.h>
#include <epicsThread.h>

static void
handler (void *arg)
{
    printf ("%s timer tripped.\n", (char *)arg);
}

int
main(int argc, char **argv)
{
    epicsTimerQueueId timerQueue;
    epicsTimerId first, second;

    /*
     * Create the queue of timer requests
     */
    timerQueue = epicsTimerQueueAllocate(1,epicsThreadPriorityScanHigh);

    /*
     * Create the timers
     */
    first = epicsTimerQueueCreateTimer(timerQueue, handler, "First");
    second = epicsTimerQueueCreateTimer(timerQueue, handler, "Second");

    /*
     * Start a timer
     */
    printf("First timer should trip in 3 seconds.\n");
    epicsTimerStartDelay(first, 3.0);
    epicsThreadSleep(5.0);
    printf("First timer should have tripped by now.\n");

    /*
     * Try starting and then cancelling a request
     */
    printf("Second timer should trip in 3 seconds.\n");
    epicsTimerStartDelay(first, 3.0);
    epicsTimerStartDelay(second, 3.0);
    epicsThreadSleep(1.0);
    epicsTimerCancel(first);
    epicsThreadSleep(5.0);
    printf("Second timer should have tripped, first timer should not have
tripped.\n");

    /*
     * Clean up a single timer
     */
```

```

epicsTimerQueueDestroyTimer(timerQueue, first);

/*
 * Clean up an entire queue of timers
 */
epicsTimerQueueRelease(timerQueue);
return 0;

```

19.12 *fdmgr*

File Descriptor Manager. `fdManager.h` describes a C++ implementation. `fdmgr.h` describes a C implementation. Neither is currently documented.

19.13 *freeList*

`freeList.h` describes routines to allocate and free fixed size memory elements. Free elements are maintained on a free list rather than being returned to the heap via calls to `free`. When it is necessary to call `malloc()`, memory is allocated in multiples of the element size.

```

void freeListInitPvt(void **ppvt, int size, int nmalloc);
void *freeListCalloc(void *pvt);
void *freeListMalloc(void *pvt);
void freeListFree(void *pvt, void *pmem);
void freeListCleanup(void *pvt);
size_t freeListItemsAvail(void *pvt);

```

where

pvt - For internal use by the freelist library. Caller must provide storage for a "void *pvt"
size - Size in bytes of each element. Note that all elements must be same size
nmalloc - Number of elements to allocate when regular `malloc()` must be called.

19.14 *gpHash*

`gpHash.h` describes a general purpose hash table for character strings. The hash table contains *tableSize* entries. Each entry is a list of members that hash to the same value. The user can maintain separate directories which share the same table by having a different *pvt* value for each directory.

```

typedef struct{
    ELLNODE    node;
    const char *name;           /*address of name placed in directory*/
    void      *pvtid;          /*private name for subsystem user*/
    void      *userPvt;        /*private for user*/
} GPENTRY;

/*tableSize must be power of 2 in range 256 to 65536*/
void gpHashInitPvt(void **ppvt,int tableSize);

```

```

GPENTRY *gphFind(void *pvt,const char *name,void *pvtid);
GPENTRY *gphAdd(void *pvt,const char *name,void *pvtid);
void gphDelete(void *pvt,const char *name,void *pvtid);
void gphFreeMem(void *pvt);
void gphDump(void *pvt);
void gphDumpFP(FILE *fp,void *pvt);

```

where

pvt - For internal use by the gpHash library. Caller must provide storage for a "void *pvt"
name - The character string that will be hashed and added to table.
pvtid - The name plus value of this pointer constitute a unique entry.

19.15 logClient

Together with the program iocLogServer this provides generic support for logging text messages from an IOC or other program to a file on the log server host machine.

A log client runs on the IOC. It accepts string messages and forwards them over a TCP connection to its designated log server (normally running on a host machine).

A log server accepts connections from multiple clients and writes the messages it receives into a rotating file. A log server program ('iocLogServer') is also part of EPICS base.

Configuration of the iocLogServer, as well as the standard iocLogClient that internally uses this library, are described in Section 10.7 on page 163.

The header file logClient.h exports the following types and routines:

```
typedef void *logClientId;
```

An abstract data type, representing a log client.

```
logClientId logClientCreate (
    struct in_addr server_addr, unsigned short server_port);
```

Create a new log client. Will block the calling task for a maximum of 2 seconds trying to connect to a server with the given ip address and port. If a connection cannot be established, an error message is printed on the console, but the log client will keep trying to connect in the background. This is done by a background task, that will also periodically (every 5 seconds) flush pending messages out to the server.

```
void logClientSend (logClientId id, const char *message);
```

Send the given message to the given log client. Messages are not immediately sent to the log server. Instead they are sent whenever the cache overflows, or logClientFlush() is called.

```
void logClientFlush (logClientId id);
```

Immediately send all outstanding messages to the server.

```
void logClientShow (logClientId id, unsigned level);
```

Print information about the log clients internal state to stdout.

For backward compatibility with older versions of the logClient library, the header file also includes iocLog.h, which exports the definitions for the standard iocLogClient for error logging. See Chapter 10.7.2.

Also for backward compatibility, the following deprecated routines are exported.

```
logClientId logClientInit (void);
```

Create a log client that uses the standard ioc log client environment variables (EPICS_IOC_LOG_INET and EPICS_IOC_LOG_PORT) as input to logClientCreate and also registers the log client with the errlog task using errlogAddListener.

```
void logClientSendMessage (logClientId id, const char *message);
```

Check the global variable iocLogDisable before calling logClientSend.

19.16 macLib

macLib.h describes a general purpose macro substitution library. It is used for all macro substitution in base.

```
long macCreateHandle(
    MAC_HANDLE **handle,          /* address of variable to receive pointer */
                                /* to new macro substitution context */
    char *pairs[]                /* pointer to NULL-terminated array of */
                                /* {name,value} pair strings; a NULL */
                                /* value implies undefined; a NULL */
                                /* argument implies no macros */
);

void macSuppressWarning(
    MAC_HANDLE *handle,          /* opaque handle */
    int falseTrue               /* 0 means issue, 1 means suppress */
);

/*following returns #chars copied, <0 if any macros are undefined*/
long macExpandString(
    MAC_HANDLE *handle,          /* opaque handle */
    char *src,                  /* source string */
    char *dest,                 /* destination string */
    long maxlen                  /* maximum number of characters to copy */
                                /* to destination string */
);

/*following returns length of value */
long macPutValue(
    MAC_HANDLE *handle,          /* opaque handle */
    char *name,                 /* macro name */
    char *value                  /* macro value */
);

/*following returns #chars copied (<0 if undefined) */
long macGetValue(
    MAC_HANDLE *handle,          /* opaque handle */
    char *name,                 /* macro name or reference */
    char *value,                /* string to receive macro value or name */
                                /* argument if macro is undefined */
    long maxlen                  /* maximum number of characters to copy */
                                /* to value */
);
```

```

long macDeleteHandle(MAC_HANDLE *handle);
long macPushScope(MAC_HANDLE *handle);
long macPopScope(MAC_HANDLE *handle);
long macReportMacros(MAC_HANDLE *handle);

/* Function prototypes (utility library) */

/*following returns #defns encountered; <0 = ERROR */
long macParseDefns(
    MAC_HANDLE *handle,          /* opaque handle; can be NULL if default */
                                /* special characters are to be used */
    char        *defns,          /* macro definitions in "a=xxx,b=yyy" */
                                /* format */
    char        **pairs[]        /* address of variable to receive pointer */
                                /* to NULL-terminated array of {name, */
                                /* value} pair strings; all storage is */
                                /* allocated contiguously */
);

/*following returns #macros defined; <0 = ERROR */
long macInstallMacros(MAC_HANDLE *handle,
    char        *pairs[]         /* pointer to NULL-terminated array of */
                                /* {name,value} pair strings; a NULL */
                                /* value implies undefined; a NULL */
                                /* argument implies no macros */
);

/*Expand string using environment variables as macro definitions */
epicsShareFunc char *           /* expanded string; NULL if any undefined macros */
epicsShareAPI macEnvExpand(
    char *str                     /* string to be expanded */
);

```

NOTE: The directory <base>/src/libCom/macLib contains two files macLibNOTES and macLibREADME that explain this library.

19.17 misc

19.17.1 aToIPAddr

The function prototype for this routine appears in `osiSock.h`

```
int aToIPAddr(const char *pAddrString, unsigned short defaultPort,
              struct sockaddr_in *pIP);
```

`aToIPAddr()` fills in the structure pointed to by the *pIP* argument with the Internet address and port number specified by the *pAddrString* argument.

Three forms of *pAddrString* are accepted:

1. n.n.n.n:p
The Internet address of the host, specified as four numbers separated by periods.
2. xxxxxxxx:p
The Internet address number of the host, specified as a single number.
3. hostname:p
The Internet host name of the host.

In all cases the ‘:p’ may be omitted in which case the port number is set to the value of the *defaultPort* argument. All numbers are read in base 16 if they begin with ‘0x’ or ‘0X’, in base 8 if they begin with ‘0’, and in base 10 otherwise.

19.17.2 adjustment

`adjustment.h` describes a single function:

```
size_t adjustToWorstCaseAlignment(size_t size);
```

`adjustToWorstCaseAlignment()` returns a value \geq *size* that an exact multiple of the worst case alignment for the architecture on which the routine is executed.

19.17.3 cantProceed

`cantProceed.h` describes routines that are provided for code that can't proceed when an error occurs.

```
void cantProceed(const char *errorMessage);  
void *callocMustSucceed(size_t count, size_t size, const char *errorMessage);  
void *mallocMustSucceed(size_t size, const char *errorMessage);
```

`cantProceed()` issues the error message and suspends the current task - it will never return. `callocMustSucceed()` and `mallocMustSucceed()` can be used in place of `calloc()` and `malloc()`. If *size* or *count* are zero, or the memory allocation fails, they output a message and call `cantProceed()`.

19.17.4 dbDefs

`dbDefs.h` contains definitions that are still used in base but should not be. Hopefully these all go away some day. This has been the hope for about ten years.

19.17.5 epicsConvert

`epicsConvert.h` currently describes:

```
float epicsConvertDoubleToFloat(double value);
```

`epicsConvertDoubleToFloat` converts a double to a float. If the double value can not be represented as a float then the assigned value is `+-FLTMIN` or `+-FLT_MAX`. A floating exception is never raised.

19.17.6 epicsString

`epicsString.h` currently describes:

```
int dbTranslateEscape(char *dst, const char *src);  
int epicsStrCaseCmp(const char *s1, const char *s2);  
int epicsStrnCaseCmp(const char *s1, const char *s2, int n);
```

```

char *epicsStrDup(const char *s);
int epicsStrPrintEscaped(FILE *fp, const char *s, int n);
int epicsStrSnPrintEscaped(char *outbuf, int outsize,
                           const char *inbuf, int inlen);
int epicsStrGlobMatch(const char *str, const char *pattern);
char * epicsStrtok_r(char *s, const char *delim, char **lasts);

```

dbTranslateEscape copies the string *src* to *dst* while substituting escape sequences. It returns the length of the resultant string (which may contain null bytes). The caller must ensure that the buffer *dst* is large enough, but the destination string can never be longer than the source and it is legal for *dst* and *src* to be the same and thus perform translations in-place.

epicsStrCaseCmp and epicsStrnCmp implement strcasecmp and strncasecmp, respectively, which are not available on all operating systems. They operate like strcmp and strncmp, but are case insensitive.

epicsStrDup implements strdup, which is not available on all operating systems. It allocates sufficient memory for a copy of the string *s*, does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(). If insufficient memory is available cantProceed() is called.

epicsStrPrintEscaped prints the contents of buffer substituting C-style escape sequences for non printable characters. In a sense it is the opposite of dbTranslateEscape. epicsStrSnPrintEscaped performs the same substitutions but stores the result in the specified output buffer. It returns the number of characters stored in the output buffer or a value greater than or equal to outsize if the output has been truncated to fit in the specified size. The following non-printable characters are printed as C-style character constants:

```
'\a', '\b', '\f', '\n', '\r', '\t', '\v', '\\', '\'', '\"'
```

All other non-printable characters are printed as:

```
'\ooo'
```

Where ooo comprises three octal digits (0-7).

epicsStrGlobMatch returns non-zero if the *str* matches the shell wild-card *pattern*.

epicsStrtok_r implements strtok_r, which is not available on all operating systems.

19.17.7 epicsTypes

epicsTypes.h provides typedefs for architecture independent data types.

```

typedef char          epicsInt8;
typedef unsigned char epicsUInt8;
typedef short        epicsInt16;
typedef unsigned short epicsUInt16;
typedef epicsUInt16  epicsEnum16;
typedef int          epicsInt32;
typedef unsigned     epicsUInt32;
typedef float        epicsFloat32;
typedef double       epicsFloat64;
typedef unsigned long epicsIndex;
typedef epicsInt32   epicsStatus;

```

So far the definitions provided in this header file have worked on all architectures. In addition to the above definitions epicsTypes.h has a number of definitions for displaying the types and other useful definitions. See the header file for details.

19.17.8 locationException

A C++ template for use as an exception object, used inside Channel Access. Not documented here.

19.17.9 shareLib.h

This is the header file for the "decorated names" that appear in header files, e.g.

```
epicsShareFunc int epicsShareAPI a_func (int arg)
```

This is used for creating DLLs for windows. Hopefully a way can be found to generated win32 DLLs which does not require decorated names.

19.17.10 truncateFile.h

```
enum TF_RETURN {TF_OK=0, TF_ERROR=1};  
TF_RETURN truncateFile (const char *pFileName, unsigned size);
```

where

pFileName - name (and optionally path) of file

truncateFile() truncates the file to the specified size. truncate() is not used because it is not portable. It returns TF_OK if the file is less than size bytes or if it was successfully truncated. It returns TF_ERROR if the file could not be truncated.

19.17.11 unixFileName.h

Defines macros OSI_PATH_LIST_SEPARATOR and OSI_PATH_SEPARATOR

19.17.12 epicsUnitTest.h

The unit test routines make it easy for a test program to generate output that is compatible with the Test Anything Protocol and can thus be used with Perl's automated Test::Harness as well as being human-readable. The routines detect whether they are being run automatically and print a summary of the results at the end if not.

```
void testPlan(int tests);  
int testOk(int pass, const char *fmt, ...);  
#define testOk1(cond) testOk(cond, "%s", #cond)  
void testPass(const char *fmt, ...);  
void testFail(const char *fmt, ...);  
int testOkV(int pass, const char *fmt, va_list pvar);  
void testSkip(int skip, const char *why)  
void testTodoBegin(const char *why);  
void testTodoEnd();  
int testDiag(const char *fmt, ...);  
void testAbort(const char *fmt, ...);  
int testDone(void);
```

A test program starts with a call to testPlan(), announcing how many tests are to be conducted. If this number is not known a value of zero can be used during development, but it is recommended that the correct value be substituted after the test program has been completed.

Individual test results are reported using any of `testOk()`, `testOk1()`, `testOkV()`, `testPass()` or `testFail()`. The `testOk()` call takes and also returns a logical pass/fail result (zero means failure, any other value is success) and a printf-like format string and arguments which describe the test. The convenience macro `testOk1()` is provided which stringifies its single condition argument, reducing the effort needed to write test programs. The individual `testPass()` and `testFail()` routines can be used when the test program takes a different path on success than on failure. The `testOkV()` routine is a varargs form of `testOk()` included for internal purposes which may prove useful in some cases.

If some program condition or failure makes it impossible to run some tests, the `testSkip()` routine can be used to indicate how many tests are being omitted from the run, thus keeping the test counts correct; the constant string `why` is displayed as an explanation to the user (this string is not printf-like).

If some tests are expected to fail because functionality in the module under test has not yet been implemented, these tests should still be executed, wrapped between calls to `testTodoBegin()` (which takes a constant string indicating why these tests are not expected to succeed) and `testTodoEnd()`. This modifies the counting of the results so these tests will not be recorded as failures.

Additional information can be supplied using the `testDiag()` routine, which displays the relevant information as a comment in the result output. None of the strings passed to any `testXxx()` routine should contain a newline `'\n'` character; newlines will be added by the test routines as part of the Test Anything Protocol. For multiple lines of diagnostic output, call `testDiag()` as many times as necessary.

If at any time the test program is unable to continue for some catastrophic reason, calling `testAbort()` with an appropriate message will ensure that the test harness understands this; `testAbort()` does not return, but calls the ANSI C routine `abort()` to cause the program to stop immediately.

After all of the tests have been completed, the return value from `testDone()` can be used as the return status code from the program's `main()` routine.

The following is a simple example of a test program using the `epicsUnitTest` routines:

```
#include <math.h>
#include "epicsUnitTest.h"

int main(int argc, char **argv)
{
    testPlan(3);
    testOk(sin(0.0) == 0.0, "Sine starts");
    testOk(cos(0.0) == 1.0, "Cosine continues");
    if (!testOk1(M_PI == 4.0*atan(1.0)))
        testDiag("4*atan(1) = %g", 4.0*atan(1.0));
    return testDone();
}
```

The output from running the above program looks like this:

```
1..3
ok 1 - Sine starts
ok 2 - Cosine continues
ok 3 - M_PI == 4.0*atan(1.0)

Results
=====
    Tests: 3
    Passed: 3 = 100%
```

Chapter 20: *libCom OSI libraries*

20.1 Overview

Most code in base is operating system independent, i.e. the code is exactly the same for all supported operating systems. This is accomplished by providing epics defined libraries for facilities that are different on the various systems. The code is called Operating System Independent or OSI. OSI libraries have multiple implementations, which are Operating System Dependent or OSD.

20.1.1 OSI source directory

Directory `<base>/src/libCom/osi` contains the code for the operating system independent libraries. The structure of this directory is:

```
osi/  
  epics*.h  
  *.cpp - A few generic c++ implementations  
  os/  
    Linux/  
    Darwin/  
    RTEMS/  
    WIN32/  
    default/  
    posix/  
    solaris/  
    vxWorks/
```

Code for additional operating systems may also be present.

20.1.2 Rules for building OSI code

The `osi` directory contains header files with names starting with `epics`. These headers provide definitions for user code. Each of the directories under `osi/<arch>` contain architecture-specific code in filenames starting with `osd`. In most cases both a header and source file are present.

Installing header files residing under `src/libCom/osi` into `<base>/include`

- Header files in `osi` are installed into `<base>/include`
- Header files in a subdirectory below `osi/os` are installed into `<base>/include/os/<arch>`. The search order for locating the specific file to be installed is:
 - `libCom/osi/os/<arch>`
 - `libCom/osi/os/posix`
 - `libCom/osi/os/default`

The search order for locating `osd` source files is:

- libCom/osi/os/<arch>
- libCom/osi/os/posix
- libCom/osi/os/default

20.1.3 Locating OSI header files.

When code is compiled the search order for locating header files in base/include is:

- <base>/include/os/<arch>
- <base>/include

20.2 *epicsAssert*

This is a replacement for ANSI C's assert. To use this version just include:

```
include "epicsAssert.h"
```

instead of

```
include <assert.h>
```

If an assert fails, it calls `errlog` indicating the program's author, file name, and line number. Under each OS there are specialized instructions assisting the user to diagnose the problem and generate a good bug report. For instance, under vxWorks, there are instructions on how to generate a stack trace, and on posix there are instructions about saving the core file. After printing the message the calling thread is suspended.

20.3 *epicsEvent*

`epicsEvent.h` contains a C++ and a C description for an event semaphore.

20.3.1 C++ Interface

```
typedef enum {
    epicsEventWaitOK, epicsEventWaitTimeout, epicsEventWaitError
} epicsEventWaitStatus;

typedef enum { epicsEventEmpty, epicsEventFull } epicsEventInitialState;

class epicsEvent {
public:
    epicsEvent(epicsEventInitialState initial=epicsEventEmpty);
    ~epicsEvent();
    void signal();
    void wait(); /*blocks until full*/
    bool wait( double timeOut ); /* false if empty at time out */
    bool tryWait(); /* false if empty */
    void show( unsigned level ) const;
```

```

class invalidSemaphore {}; /* exception */
private:
    ...
};

```

Method	Meaning
epicsEvent	An epicsEvent can be created empty or full. If it is created empty then a wait issued before a signal will block. If created full then the first wait will always succeed. Multiple signals may be issued between waits but have the same effect as a single signal.
~epicsEvent	Remove the event and any resources it uses. Any further use of the semaphore result in unknown (most certainly bad) behavior. No outstanding take can be active when this call is made.
signal	Signal the event i.e. ensures that the next or current call to wait completes. This method may be called from a vxWorks or RTEMS interrupt handler.
wait()	Wait for the event.
wait(double timeOut)	Similar to wait except that if event does not happen the call completes after the specified time out. The return value is (false,true) if the event (did not, did) happen.
tryWait()	Similar to wait except that if event does not happen the call completes immediately. The return value is (false,true) if the event (did not, did) happen.
show	Display information about the semaphore. The information displayed is architecture dependent.

The primary use of an event semaphore is for synchronization. An example of using an event semaphore is a consumer thread that processes requests from one or more producer threads. For example:

- When creating the consumer thread also create an epicsEvent.

```
epicsEvent *pevent = new epicsEvent;
```
- The consumer thread has code containing:

```

while(1) {
    pevent->wait();
    while(/*more work*/) {
        /*process work*/
    }
}

```
- Producers create requests and issue the statement:

```
pevent->signal();
```

20.3.2 C Interface

```

typedef struct epicsEventOSD *epicsEventId;

epicsEventId epicsEventCreate(epicsEventInitialState initialState);
epicsEventId epicsEventMustCreate (epicsEventInitialState initialState);
void epicsEventDestroy(epicsEventId id);

```

```

void epicsEventSignal(epicsEventId id);
epicsEventWaitStatus epicsEventWait(epicsEventId id);
void epicsEventMustWait(epicsEventId id);
epicsEventWaitStatus epicsEventWaitWithTimeout(epicsEventId id, double timeOut);
epicsEventWaitStatus epicsEventTryWait(epicsEventId id);
void epicsEventShow(epicsEventId id, unsigned int level);

```

Each C routine corresponds to one of the C++ methods. `epicsEventMustCreate` and `epicsEventMustWait` do not return if they fail.

20.4 *epicsFindSymbol*

`epicsFindSymbol.h` contains the following definition:

```
void * epicsFindSymbol(const char *name);
```

Method	Meaning
<code>epicsFindSymbol</code>	Return the address of the variable name

`vxWorks` provides a function `symFindByName`, which finds and returns the address of global variables. The registry, described in the next chapter, provides an alternative but also requires extra work by `iocCore` and/or user code. If the registry is asked for a name that has not been registered, it calls `epicsFindSymbol`. If `epicsFindSymbol` can locate the global symbol it returns the address, otherwise it returns null.

On `vxWorks` `epicsFindSymbol` calls `symFindByName`.

A default version just returns null, i.e. it always fails.

20.5 *epicsInterrupt*

`epicsInterrupt.h` contains the following:

20.5.1 C Interface

```

int epicsInterruptLock();
void epicsInterruptUnlock(int key);
int epicsInterruptIsInterruptContext();
void epicsInterruptContextMessage(const char *message);

```

Method	Meaning
epicsInterruptLock	Lock interrupts and return a key to be passed to epicsInterruptUnlock To lock the following is done. int key; ... key = epicsInterruptLock(); ... epicsInterruptUnlock(key);
epicsInterruptUnlock	Unlock interrupts.
epicsInterruptIsInterruptContext	Return (true, false) if current context is interrupt context.
epicsInterruptContextMessage	Generate a message while interrupt context is true.

20.5.2 Implementation notes

A vxWorks specific version is provided. It maps directly to intLib calls.

An RTEMS version is provided that maps to rtems_ calls.

A default version is provided that uses a global semaphore to lock. This version is intended for operating systems in which iocCore will run as a multi threaded process. The global semaphore is thus only global within the process. This version is intended for use on all except real time operating systems.

The vxWorks implementation will most likely not work on symmetric multiprocessing systems.

The reason epicsInterrupt is needed is:

- callbackRequest and scanOnce can be issued from interrupt level.
- The errlog routines can be called while at interrupt level.

20.6 epicsMath

epicsMath.h includes math.h and also ensures that isnan and isinf are defined.

20.7 epicsMessageQueue

epicsMessageQueue.h describes a C++ and a C facility for interlocked communication between threads.

20.7.1 C++ Interface

EpicsMessageQueue provides methods for sending messages between threads on a first in, first out basis. It is designed so that a message queue can be used with multiple writer and reader threads

```

class epicsMessageQueue {
public:
    epicsMessageQueue(unsigned int capacity, unsigned int maximumMessageSize);
    ~epicsMessageQueue();
    bool trySend(void *message, unsigned int messageSize);
    bool send(void *message, unsigned int messageSize);
    bool send(void *message, unsigned int messageSize, double timeout);
    int tryReceive(void *message, unsigned int messageBufferSize);
    int receive(void *message, unsigned int messageBufferSize);
    int receive(void *message, unsigned int messageBufferSize, double timeout);
    void show(int level) const;
    int pending() const;

private: // Prevent compiler-generated member functions
    // default constructor, copy constructor, assignment operator
    epicsMessageQueue();
    epicsMessageQueue(const epicsMessageQueue &);
    epicsMessageQueue& operator=(const epicsMessageQueue &);

private: // Data
    ...
};

```

An `epicsMessageQueue` cannot be assigned to, copy-constructed, or constructed without giving the *capacity* and *maximumMessageSize* arguments. The C++ compiler will object to some of the statements below:

```

epicsMessageQueue mq0(); // Error: default constructor is private
epicsMessageQueue mq1(10, 20); // OK
epicsMessageQueue mq2(t1); // Error: copy constructor is private
epicsMessageQueue *pmq; // OK, pointer
*pmq = mq1; // Error: assignment operator is private
pmq = &mq1; // OK, pointer assignment and address-of

```

Method	Meaning
<code>epicsMessageQueue()</code>	Constructor. The capacity is the maximum number of messages, each containing 0 to <code>maximumMessageSize</code> bytes, that can be stored in the message queue.
<code>~epicsMessageQueue()</code>	Destructor.
<code>trySend()</code>	Try to send a message. Return 0 if the message was sent to a receiver or queued for future delivery. Return -1 if no more messages can be queued or if the message is larger than the queue's maximum message size. This method may be called from a vxWorks or RTEMS interrupt handler.
<code>send()</code>	Send a message. Return 0 if the message was sent to a receiver or queued for future delivery. Return -1 if the timeout, if any, was reached before the message could be sent or queued, or if the message is larger than the queue's maximum message size.
<code>tryReceive()</code>	Try to receive a message. If the message queue is non-empty, the first message on the queue is copied to the specified location and the length of the message is returned. Returns -1 if the message queue is empty. If the pending message is larger than the specified <code>messageBufferSize</code> it may either return -1, or truncate the message. It is most efficient if the <code>messageBufferSize</code> is equal to the <code>maximumMessageSize</code> with which the message queue was created.

Method	Meaning
receive()	Wait until a message is sent and store it in the specified location. The number of bytes in the message is returned. Returns -1 if a message is not received within the timeout interval. If the received message is larger than the specified messageBufferSize it may either return -1, or truncate the message. It is most efficient if the messageBufferSize is equal to the maximumMessageSize with which the message queue was created.
show()	Displays some information about the message queue. The level argument controls the amount of information displayed.
pending()	Returns the number of messages presently in the queue.

20.7.2 C interface

```
typedef void *epicsMessageQueueId;
epicsMessageQueueId epicsMessageQueueCreate(unsigned int capacity,
                                             unsigned int maximumMessageSize);
void epicsMessageQueueDestroy(epicsMessageQueueId);
int epicsMessageQueueTrySend(epicsMessageQueueId, void *, unsigned int);
int epicsMessageQueueSend(epicsMessageQueueId, void *, unsigned int);
int epicsMessageQueueSendWithTimeout(epicsMessageQueueId, void *, unsigned int,
                                      double);
int epicsMessageQueueTryReceive(epicsMessageQueueId, void *, unsigned int);
int epicsMessageQueueReceive(epicsMessageQueueId, void *, unsigned int);
int epicsMessageQueueReceiveWithTimeout(epicsMessageQueueId, void*,
                                         unsigned int, double);
void epicsMessageQueueShow(epicsMessageQueueId);
int epicsMessageQueuePending(epicsMessageQueueId);
```

Each C function corresponds to one of the C++ methods.

20.8 *epicsMutex*

epicsMutex.h contains both C++ and C descriptions for a mutual exclusion semaphore.

20.8.1 C++ Interface

```
typedef enum {
    epicsMutexLockOK, epicsMutexLockTimeout, epicsMutexLockError
} epicsMutexLockStatus;
class epicsMutex {
public:
    epicsMutex ();
    ~epicsMutex ();
    void lock (); /* blocks until success */
    bool tryLock (); /* true if successful */
    void unlock ();
    void show ( unsigned level ) const;
```

```

    class invalidSemaphore {}; /* exception */
private:
};

```

Method	Meaning
epicsMutex	Create a mutual exclusion semaphore.
~epicsMutex	Remove the semaphore and any resources it uses. Any further use of the semaphore result in unknown (most certainly bad) results.
lock()	Wait until the resource is free. After a successful lock additional, i.e. recursive, locks of any type can be issued but each must have an associated unlock.
tryLock()	Similar to lock except that, if the resource is owned by another thread, the call completes immediately. The return value is (false,true) if the resource (is not, is) owned by the caller.
unlock	Release the resource. If a thread issues recursive locks, there must be an unlock for each lock
show	Display information about the semaphore. The results are architecture dependent.

Mutual exclusion semaphores are for situations requiring mutually exclusive access to resources. A mutual exclusion semaphore may be taken recursively, i.e. can be taken more than once by the owner thread before releasing it. Recursive takes are useful for a set of routines that call each other while working on a mutually exclusive resource.

The typical use of a mutual exclusion semaphore is:

```

epicsMutex *plock = new epicsMutex;
...
...
plock->lock();
/* process resource */
plock->unlock();

```

20.8.2 C Interface

```

typedef struct epicsMutexOSD* epicsMutexId;

epicsMutexId epicsMutexCreate(void);
epicsMutexId epicsMutexMustCreate (void);
void epicsMutexDestroy(epicsMutexId id);
void epicsMutexUnlock(epicsMutexId id);
epicsMutexLockStatus epicsMutexLock(epicsMutexId id);
void epicsMutexMustLock(epicsMutexId id);
epicsMutexLockStatus epicsMutexTryLock(epicsMutexId id);
void epicsMutexShow(epicsMutexId id,unsigned int level);
void epicsMutexShowAll(int onlyLocked,unsigned int level);

```

Each C routine corresponds to one of the C++ methods. `epicsMutexMustCreate` and `epicsMutexMustLock` do not return if they fail.

20.8.3 Implementation Notes

The implementation:

- Must implement recursive locking
- May implement priority inheritance and be deletion safe

A posix version is implemented via pthreads.

20.9 *epicsStdlib*

epicsStdlib.h includes stdlib.h and contains definitions for the following functions/macros.

```
double epicsStrtod(const char *str, char **endp);
int epicsScanFloat(const char *str, float *dest);
int epicsScanDouble(const char *str, double *dest);
```

epicsStrtod has the same semantics as the C99 function strtod. It is provided because some architectures have implementations which do not accept NAN or INFINITY. On architectures which provide an acceptable version of strtod this is implemented as a simple macro expansion.

epicsScanFloat and epicsScanDouble behave like sscanf with a "%f" and "%lf" format string, respectively. They are provided because some architectures have implementations of scanf which do not accept NAN or INFINITY.

20.10 *epicsStdio*

epicsStdio.h contains definitions for the following functions.

```
int epicsSnprintf(char *str, size_t size,
    const char *format, ...);
int epicsVsnprintf(char *str, size_t size,
    const char *format, va_list ap);
void epicsTempName(char * pNameBuf, size_t nameBufLength );
FILE * epicsShareAPI epicsTempFile ();
enum TF_RETURN {TF_OK=0, TF_ERROR=1};
enum TF_RETURN truncateFile(const char *pFileName, unsigned size );
FILE * epicsGetStdin(void);
FILE * epicsGetStdout(void);
FILE * epicsGetStderr(void);
FILE * epicsGetThreadStdin(void);
FILE * epicsGetThreadStdout(void);
FILE * epicsGetThreadStderr(void);
void epicsSetThreadStdin(FILE *);
void epicsSetThreadStdout(FILE *);
void epicsSetThreadStderr(FILE *);
int epicsStdoutPrintf(const char *pformat, ...);
```

`epicsSnprintf` and `epicsVsnprintf` are meant to have the same semantics as the C99 functions `snprintf` and `vsnprintf`. They are provided because some architectures do not implement these functions. `snprintf` is like `sprintf` except that the `size` argument limits the number of characters placed in `str`. Similarly `vsnprintf` is like `vsprintf`.

The WIN32 implementation does not provide correct C99 semantics for the return value. On WIN32 `epicsSnprintf` and `epicsVsnprintf` return `size` when the output message is truncated rather than the number of characters that would have been printed if the size were unlimited (not including the final `'\0'`).

Operating systems such as Solaris, which follow the Single Unix Specification V2, do not provide correct C99 semantics for the return value. On these systems `epicsSnprintf` and `epicsVsnprintf` return a value less than 0 when a buffer length of 0 is passed.

`epicsTempName` and `epicsTempFile` can be called to get unique filenames and files.

`truncateFile` returns `TF_OK` if the file is less than `size` bytes or if it was successfully truncated. Returns `TF_ERROR` if the file could not be truncated.

The `Stdin/Stdout/Stderr` routines allow these file streams to be redirected on a per thread basis, e.g. calling `epicsSetThreadStdout` will affect only the thread which calls it.

`epicsGetStdin`, ..., `epicsStdoutPrintf` are not normally called by user code. Instead code that wants to allow redirection needs only to include `epicsStdioRedirect.h`

20.11 epicsStdioRedirect

Including this file cause the following names to be changed:

- `stdin` becomes `epicsGetStdin()`
- `stdout` becomes `epicsGetStdout()`
- `stderr` becomes `epicsGetStderr()`
- `printf` becomes `epicsStdoutPrintf`

This is done so that redirection can occur. A primary use of this facility is `iocsh`. It allows redirection of input and output. In order for it to work, all modules involved in I/O can just include this header.

20.12 epicsThread

`epicsThread.h` contains C++ and C descriptions for a thread.

20.12.1 C Interface

```
typedef void (*EPICSTHREADFUNC)(void *parm);

static const unsigned epicsThreadPriorityMax = 99;
static const unsigned epicsThreadPriorityMin = 0;
/* some generic values */
static const unsigned epicsThreadPriorityLow = 10;
static const unsigned epicsThreadPriorityMedium = 50;
static const unsigned epicsThreadPriorityHigh = 90;
```

```
/* some iocCore specific values */
static const unsigned epicsThreadPriorityCAServerLow = 20;
static const unsigned epicsThreadPriorityCAServerHigh = 40;
static const unsigned epicsThreadPriorityScanLow = 60;
static const unsigned epicsThreadPriorityScanHigh = 70;
static const unsigned epicsThreadPriorityIocsh = 91;
static const unsigned epicsThreadPriorityBaseMax = 91;

/* stack sizes for each stackSizeClass are implementation and CPU dependent */
typedef enum {
    epicsThreadStackSmall, epicsThreadStackMedium, epicsThreadStackBig
} epicsThreadStackSizeClass;

typedef enum {
    epicsThreadBooleanStatusFail, epicsThreadBooleanStatusSuccess
} epicsThreadBooleanStatus;

unsigned int epicsThreadGetStackSize(epicsThreadStackSizeClass size);

typedef int epicsThreadOnceId;
#define EPICS_THREAD_ONCE_INIT 0

void epicsThreadOnce(epicsThreadOnceId *id, EPICSTHREADFUNC, void *arg);

void epicsThreadExitMain(void);

/* (epicsThreadId)0 is guaranteed to be an invalid thread id */
typedef struct epicsThreadOSD *epicsThreadId;

epicsThreadId epicsThreadCreate(const char *name,
    unsigned int priority, unsigned int stackSize,
    EPICSTHREADFUNC funptr, void *parm);
void epicsThreadSuspendSelf(void);
void epicsThreadResume(epicsThreadId id);
unsigned int epicsThreadGetPriority(epicsThreadId id);
unsigned int epicsThreadGetPrioritySelf();
void epicsThreadSetPriority(epicsThreadId id, unsigned int priority);
epicsThreadBooleanStatus epicsThreadHighestPriorityLevelBelow (
    unsigned int priority, unsigned *pPriorityJustBelow);
epicsThreadBooleanStatus epicsThreadLowestPriorityLevelAbove (
    unsigned int priority, unsigned *pPriorityJustAbove);
int epicsThreadIsEqual(epicsThreadId id1, epicsThreadId id2);
int epicsThreadIsSuspended(epicsThreadId id);
void epicsThreadSleep(double seconds);
double epicsThreadSleepQuantum(void);
epicsThreadId epicsThreadGetIdSelf(void);
epicsThreadId epicsThreadGetId(const char *name);

const char * epicsThreadGetNameSelf(void);
void epicsThreadGetName(epicsThreadId id, char *name, size_t size);
int epicsThreadIsOkToBlock(void);
void epicsThreadSetOkToBlock(int isOkToBlock);
```

```

void epicsThreadShowAll(unsigned int level);
void epicsThreadShow(epicsThreadId id,unsigned int level);

typedef void * epicsThreadPrivateId;
epicsThreadPrivateId epicsThreadPrivateCreate(void);
void epicsThreadPrivateDelete(epicsThreadPrivateId id);
void epicsThreadPrivateSet(epicsThreadPrivateId,void *);
void * epicsThreadPrivateGet(epicsThreadPrivateId);

```

Method	Meaning
epicsThreadGetStackSize	Get a stack size value that can be given to epicsThreadCreate. The size argument should be one of the values epicsThreadStackSmall, epicsThreadStackMedium or epicsThreadStackBig.
epicsThreadOnce	This is used as follows: <pre> void myInitFunc(void * arg) { ... } epicsThreadId onceFlag = EPICS_THREAD_ONCE_INIT; ... epicsThreadOnce(&onceFlag,myInitFunc,(void *)myParm); </pre> For each unique epicsThreadId, epicsThreadOnce guarantees 1) myInitFunc is called only once. 2) myInitFunc completes before any epicsThreadOnce call completes. Note that myInitFunc must not call epicsThreadOnce with the same onceId.
epicsThreadExitMain	If the main routine is done but wants to let other threads run it can call this routine. This should be the last call in main, except the final return. On most systems epicsThreadExitMain never returns. This must only be called by the main thread.
epicsThreadCreate	Create a new thread. The use made of the priority, and stackSize arguments is implementation dependent. Some implementations may ignore one or other of these, but for portability appropriate values should be given for both. The value passed as the stackSize parameter should be obtained by calling epicsThreadGetStackSize. The funptr argument specifies a function that implements the thread, and parm is the single argument passed to funptr. A thread terminates when funptr returns.
epicsThreadSuspendSelf	This causes the calling thread to suspend. The only way it can resume is for another thread to call epicsThreadResume.
epicsThreadResume	Resume a suspended thread. Only do this if you know that it is safe to resume a suspended thread.
epicsThreadGetPriority	Get the priority of the specified thread.
epicsThreadGetPrioritySelf	Get the priority of this thread.
epicsThreadSetPriority	Set a new priority for the specified thread. The result is implementation dependent.
epicsThreadHighestPriorityLevelBelow	Get a priority that is just lower than the specified priority.
epicsThreadLowestPriorityLevelAbove	Get a priority that is just above the specified priority.

Method	Meaning
epicsThreadIsEqual	Compares two threadIds and returns (0,1) if they (are not, are) the same.
epicsThreadIsSuspended	BAD NAME. taskwd needs this call. It really means: Is there something wrong with this thread? This could mean suspended or no longer exists or etc. It is a problem because it is implementation dependent.
epicsThreadSleep	Sleep for the specified period of time, i.e. sleep without using the cpu. If delay is >0 then the thread will sleep at least until the next clock tick. The exact time is determined by the underlying architecture. If delay is <= 0 then a delay of 0 is requested of the underlying architecture. What happens is architecture dependent but often it allows other threads of the same priority to run.
epicsThreadSleepQuantum	This function returns the minimum slumber interval obtainable with epicsThreadSleep() in seconds. On most OS there is a system scheduler interrupt interval which determines the value of this parameter. Knowledge of this parameter is used by the various components of EPICS to improve scheduling of software tasks in time when the reduction of average time scheduling errors is important. If this parameter is unknown or is unpredictable for a particular OS then it is safe to return zero.
epicsThreadGetIdSelf	Get the threadId of the calling thread.
epicsThreadGetId	Get the threadId of the specified thread. A return of 0 means that no thread was found with the specified name.
epicsThreadGetNameSelf	Get the name of the calling thread.
epicsThreadGetName	Get the name of the specified thread. The value is copied to a caller specified buffer so that if the thread terminates the caller is not left with a pointer to something that may no longer exist.
epicsThreadIsOkToBlock	Is it OK for a thread to block? This can be called by support code that does not know if it is called in a thread that should not block. For example the errlog system calls this to decide when messages should be displayed on the console.
epicsThreadSetOkToBlock	When a thread is started the default is that it is not allowed to block. This method can be called to change the state. For example iocsh calls this to specify that it is OK to block.
epicsThreadShowAll	Display info about all threads.
epicsThreadShow	Display info about the specified thread.
epicsThreadPrivateCreate	Thread private variables are intended for use by legacy libraries written for a single threaded environment and which uses a global variable to store private data. The only code in base that currently needs this facility is channel access. A library that needs a private variable should make exactly one call to epicsThreadPrivateCreate. Each thread should call epicsThreadPrivateSet when the thread is created. Each library routine can call epicsThreadPrivateGet each time it is called.
epicsThreadPrivateDelete	Delete a thread private variable.
epicsThreadPrivateSet	Set the value for a thread private variable.

Method	Meaning
epicsThreadPrivateGet	Get the value of a thread private variable, the value is the value set by the call to epicsThreadPrivateSet that was made by the same thread. If called before epicsThreadPrivateSet it returns 0.

epicsThread is meant as a somewhat minimal interface for multithreaded applications. It can be implemented on a wide variety of systems with the restriction that the system **MUST** support a multithreaded environment. A POSIX pthreads version is provided.

The interface provides the following thread facilities, with restrictions as noted:

- Life cycle - A thread starts life as a result of a call to epicsThreadCreate. It terminates when the thread function returns. It should not return until it has released all resources it uses. If a thread is expected to terminate as a natural part of its life cycle then the thread function must return.
- epicsThreadOnce - This provides the ability to have an initialization function that is guaranteed to be called exactly once.
- main - If a main routine finishes its work but wants to leave other threads running it can call epicsThreadExitMain, which should be the last statement in main.
- Priorities - Ranges between 0 and 99 with a higher number meaning higher priority. A number of constants are defined for iocCore specific threads. The underlying implementation may collapse the range 0 to 99 into a smaller range; even a single priority. User code should never use priorities to guarantee correct behavior.
- Stack Size - epicsThreadCreate accepts a stack size parameter. Three generic sizes are available: small, medium, and large. Portable code should always use one of the generic sizes. Some implementation may ignore the stack size request and use a system default instead. Virtual memory systems providing generous stack sizes can be expected to use the system default.
- epicsThreadId - This is given a value as a result of a call to epicsThreadCreate. A value of 0 always means no thread. If a threadId is used for a thread that has terminated the result is not defined (but will normally lead to bad things happening). Thus code that looks after other threads **MUST** be aware of threads terminating.

20.12.2 C++ Interface

```
class epicsThreadRunnable {
public:
    virtual void run() = 0;
    virtual void stop();
    virtual void show(unsigned int level) const;
};

class epicsShareClass epicsThread {
public:
    epicsThread (epicsThreadRunnable &,const char *name, unsigned int stackSize,
                unsigned int priority=epicsThreadPriorityLow);
    virtual ~epicsThread ();
    void start();
    void exitWait ();
    bool exitWait (const double delay );
    void exitWaitRelease (); // noop if not called by managed thread
    static void exit ();
    void resume ();
    void getName (char *name, size_t size) const;
    epicsThreadId getId () const;
```

```

    unsigned int getPriority () const;
    void setPriority (unsigned int);
    bool priorityIsEqual (const epicsThread &otherThread) const;
    bool isSuspended () const;
    bool isCurrentThread () const;
    bool operator == (const epicsThread &rhs) const;
    /* these operate on the current thread */
    static void suspendSelf ();
    static void sleep (double seconds);
    static epicsThread & getSelf ();
    static const char * getNameSelf ();
    static bool isOkToBlock ();
    static void setOkToBlock(bool isOkToBlock) ;
private:
    ...
};
template <class T>
class epicsThreadPrivate {
public:
    epicsThreadPrivate ();
    ~epicsThreadPrivate ();
    T *get () const;
    void set (T *);
    class unableToCreateThreadPrivate {}; // exception
private:
    ...
};

```

The C++ interface is just a wrapper around the C interface. Two differences are the method `start` and the class `epicsThreadRunnable`.

The method `start` must be called only after the `epicsThread` object is constructed. It in turn calls the `run` method of the `epicsThreadRunnable` object.

Code using the C++ interface code must provide a class that derives from `epicsThreadRunnable`. One way to accomplish this is as follows:

```

class myThread: public epicsThreadRunnable {
public:
    myThread(int arg,const char *name);
    virtual ~myThread();
    virtual void run();
    epicsThread thread;
    ...
}

myThread::myThread(int arg,const char *name) :
    thread(*this,name,epicsThreadGetStackSize(epicsThreadStackSmall),50)
{
    thread.start();
}

myThread::~~myThread() {}

void myThread::run()

```

```
{
    ...
}
```

20.13 epicsTime

epicsTime.h contains C++ and C descriptions for time.

20.13.1 Time Related Structures

```
/* epics time stamp for C interface*/
typedef struct epicsTimeStamp {
    epicsUInt32    secPastEpoch;    /* seconds since 0000 Jan 1, 1990 */
    epicsUInt32    nsec;            /* nanoseconds within second */
} epicsTimeStamp;

/*TS_STAMP is deprecated */
#define TS_STAMP epicsTimeStamp

struct timespec; /* POSIX real time */
struct timeval; /* BSD */
struct l_fp; /* NTP timestamp */

// extend ANSI C RTL "struct tm" to include nano seconds within a second
// and a struct tm that is adjusted for the local timezone
struct local_tm_nano_sec {
    struct tm ansi_tm; /* ANSI C time details */
    unsigned long nSec; /* nano seconds extension */
};

// extend ANSI C RTL "struct tm" to includes nano seconds within a second
// and a struct tm that is adjusted for GMT (UTC)
struct gm_tm_nano_sec {
    struct tm ansi_tm; /* ANSI C time details */
    unsigned long nSec; /* nano seconds extension */
};

// wrapping this in a struct allows conversion to and
// from ANSI time_t but does not allow unexpected
// conversions to occur
struct time_t_wrapper {
    time_t ts;
};
```

The above structures are for the various time formats.

- `epicsTimeStamp` - This is the structure used by the C interface for epics time stamps. The C++ interface stores this information in private members. The two elements of the class are:
 - `secPastEpoch` - The number of seconds since January 1, 1990 (the epics epoch).

- nsec - nanoseconds within a second
- NOTE: TS_STAMP is defined for compatibility with existing code.
- timespec - This is defined by POSIX Real Time. It requires two mandatory fields:
 - time_t tv_sec - Number of seconds since 1970 (The POSIX epoch)
 - long tv_nsec - nanoseconds within a second
 - timeval - BSD and SRV5 Unix timestamp. It has two fields:
 - time_t tv_sec - Number of seconds since 1970 (The POSIX epoch)
 - time_t tv_nsec - nanoseconds within a second
 - struct l_fp - Network Time Protocol timestamp. The fields are:
 - l_ui - Number of seconds since 1900 (The NTP epoch)
 - l_uf - Fraction of a second. For example 0x80000000 represents 1/2 second.
 - local_tm_nano_sec and gm_tm_nano_sec - Defined by epics. It just adds a nanosecond field to struct tm.
 - time_t_wrapper - This is for converting to/from the ANSI C time_t. Since time_t is usually an elementary type providing a conversion operator from time_t to/from epicsTime could cause undesirable implicit conversions. Providing a conversion operator to/from a time_t_wrapper prevents implicit conversions.

NOTE on conversion. The epics implementation will properly convert between the various formats from the beginning of the EPICS epoch until at least 2038. Unless the underlying architecture support has defective POSIX, BSD/SRV5, or standard C time support the epics implementation should be valid until 2106.

20.13.2 C++ Interface

```
class epicsTime;

class epicsTimeEvent {
    friend class epicsTime;
public:
    epicsTimeEvent (const int &eventName);
private:
    int eventNumber;
};

class epicsTime {
public:
    // exceptions
    class unableToFetchCurrentTime {};
    class formatProblemWithStructTM {};

    epicsTime ();
    epicsTime (const epicsTime &t);

    static epicsTime getEvent (const epicsTimeEvent &event);
    static epicsTime getCurrent ();

    // convert to and from EPICS epicsTimeStamp format
    operator epicsTimeStamp () const;
    epicsTime (const epicsTimeStamp &ts);
    epicsTime operator = (const epicsTimeStamp &rhs);
```

```

// convert to and from ANSI time_t
operator time_t_wrapper () const;
epicsTime (const time_t_wrapper &tv);
epicsTime operator = (const time_t_wrapper &rhs);

// convert to and from ANSI Cs "struct tm" (with nano seconds)
// adjusted for the local time zone
operator local_tm_nano_sec () const;
epicsTime (const local_tm_nano_sec &ts);
epicsTime operator = (const local_tm_nano_sec &rhs);

// convert to ANSI Cs "struct tm" (with nano seconds)
// adjusted for GM time (UTC)
operator gm_tm_nano_sec () const;

// convert to and from POSIX RT's "struct timespec"
operator struct timespec () const;
epicsTime (const struct timespec &ts);
epicsTime operator = (const struct timespec &rhs);

// convert to and from BSD's "struct timeval"
operator struct timeval () const;
epicsTime (const struct timeval &ts);
epicsTime operator = (const struct timeval &rhs);

// convert to and from NTP timestamp format
operator l_fp () const;
epicsTime (const l_fp &);
epicsTime operator = (const l_fp &rhs);

// convert to and from WIN32s FILETIME (implemented only on WIN32)
operator struct _FILETIME () const;
epicsTime ( const struct _FILETIME & );
epicsTime & operator = ( const struct _FILETIME & );

// arithmetic operators
double operator- (const epicsTime &rhs) const; // returns seconds
epicsTime operator+ (const double &rhs) const; // add rhs seconds
epicsTime operator- (const double &rhs) const; // subtract rhs seconds
epicsTime operator+= (const double &rhs); // add rhs seconds
epicsTime operator-= (const double &rhs); // subtract rhs seconds

// comparison operators
bool operator == (const epicsTime &rhs) const;
bool operator != (const epicsTime &rhs) const;
bool operator <= (const epicsTime &rhs) const;
bool operator < (const epicsTime &rhs) const;
bool operator >= (const epicsTime &rhs) const;
bool operator > (const epicsTime &rhs) const;

// convert current state to user-specified string
size_t strftime (char *pBuff, size_t bufLength, const char *pFormat) const;

```

```

    // dump current state to standard out
    void show (unsigned interestLevel) const;

    // deprecated
    static void synchronize ();
private:
    ...
};

```

20.13.3 class epicsTimeEvent

```

class epicsShareClass epicsTimeEvent
{
    friend class epicsTime;
public:
    epicsTimeEvent (const int &eventName);
private:
    int eventNumber;
};

```

Method	Meaning
epicsTimeEvent(eventName)	This is the only method provided for this class. Why is it needed?

20.13.4 class epicsTime

Method	Meaning
epicsTime() epicsTime(const epicsTime& t);	The default constructor sets the time to the beginning of the epics epoch.
getEvent	Returns the time for the associated event. See the description of the C routine epicsTimeGetEvent described below for details.
getCurrent	Gets the current time. An example is: epicsTime time = epicsTime::getCurrent();

Method	Meaning
<p>convert to/from epicsTimeStamp</p>	<p>Three methods are provided for epicsTimeStamp. A copy constructor, an assignment operator, and a conversion to epicsTimeStamp. Assume the following definitions:</p> <pre>epicsTime time; epicsTimeStamp ts;</pre> <p>An example of the copy constructor is:</p> <pre>epicsTime time1(ts);</pre> <p>An example of the assignment operator is:</p> <pre>time = ts;</pre> <p>An example of the epicsTimeStamp operator is:</p> <pre>ts = time;</pre>
<p>Convert to/from ANSI time_t</p>	<p>Three methods are provided for ANSI time_t. A copy constructor, an assignment operator, and a conversion to time_t_wrapper. The structure time_t_wrapper must be used instead of time_t because undesired conversions could occur: Assume the following definitions:</p> <pre>time_t tt; time_t_wrapper ttw; epicsTime time;</pre> <p>An example of the copy constructor is:</p> <pre>ttw.tt = tt; epicsTime time1(ttw);</pre> <p>An example of the assignment operator is:</p> <pre>time = ttw;</pre> <p>An example of the time_t_wrapper operator is:</p> <pre>ttw = time; tt = ttw.tt;</pre>
<p>convert to and from tm_nano_sec</p>	<p>Three methods are provided for tm_nano_sec. A copy constructor, an assignment operator, and a conversion to tm_nano_sec. Assume the following definitions:</p> <pre>local_tm_nano_sec ttn; epicsTime time;</pre> <p>An example of the copy constructor is:</p> <pre>epicsTime time1(ttn);</pre> <p>An example of the assignment operator is:</p> <pre>time = ttn;</pre> <p>An example of the tm_nano_sec operator is:</p> <pre>ttn = time;</pre>

Method	Meaning
<p>convert to and from POSIX RT's "struct timespec"</p>	<p>Three methods are provided for struct timespec. A copy constructor, an assignment operator, and a conversion to struct timespec. Assume the following definitions:</p> <pre>struct timespec tts; epicsTime time;</pre> <p>An example of the copy constructor is:</p> <pre>epicsTime time1(tts);</pre> <p>An example of the assignment operator is:</p> <pre>time = tts;</pre> <p>An example of the struct timespec operator is:</p> <pre>tts = time;</pre>
<p>convert to and from BSD's "struct timeval"</p>	<p>Three methods are provided for struct timeval. A copy constructor, an assignment operator, and a conversion to struct timeval. Assume the following definitions:</p> <pre>struct timeval ttv; epicsTime time;</pre> <p>An example of the copy constructor is:</p> <pre>epicsTime time1(ttv);</pre> <p>An example of the assignment operator is:</p> <pre>time = ttv;</pre> <p>An example of the struct timeval operator is:</p> <pre>ttv = time;</pre>
<p>convert to and from NTP timestamp format</p>	<p>Three methods are provided for ntpTimeStamp. A copy constructor, an assignment operator, and a conversion to ntpTimeStamp. Assume the following definitions:</p> <pre>l_fp ntp; epicsTime time;</pre> <p>An example of the copy constructor is:</p> <pre>epicsTime time1(ntp);</pre> <p>An example of the assignment operator is:</p> <pre>time = ntp;</pre> <p>An example of the ntpTimeStamp operator is:</p> <pre>ntp = time;</pre>

Method	Meaning
arithmetic operators - + += -=	The arithmetic operators allow the difference of two epicsTimes, with the result in seconds. It also allows -, +, +=, and -= where the left hand argument is an epicsTime and the right hand argument is a double. Examples are: <pre> epicsTime time, time1, time2; double t1,t2,t3; ... t1 = time2 - time1; time = time1 + 4.5; time = time2 - t3; time2 += 6.0; </pre>
Comparison operators ==, !=, <=, <, >=, >	Two epics times can be compared: <pre> epicsTime time1, time2; ... if(time1<=time2) { ... </pre>
strftime	This is a facility similar to the ANSI C library routine strftime. See K&R for details about strftime. The epicsTime method also provides support for the printing the nanoseconds portion of the time. It looks in the format string for the sequence "%0nf" where <i>n</i> is the desired precision. It uses this format to convert the nanoseconds value to the requested precision. For example: <pre> epicsTime time = epicsTime::getCurrent(); char buf[30]; time.strftime(buf,30,"%Y/%m/%d %H:%M:%S.%06f"); printf("%s\n",buf); </pre> Will print the time in the format: <pre> 2001/01/26 20:50:29.813505 </pre>
show	Shows the date/time.

20.13.5 C Interface

```

/* All epicsTime routines return (-1,0) for (failure,success) */
#define epicsTimeOK 0
#define epicsTimeERROR (-1)
/*Some special values for eventNumber*/
#define epicsTimeEventCurrentTime 0
#define epicsTimeEventBestTime -1
#define epicsTimeEventDeviceTime -2

int epicsTimeGetCurrent (epicsTimeStamp *pDest);
int epicsTimeGetEvent (epicsTimeStamp *pDest, int eventNumber);

/* convert to and from ANSI C's "time_t" */
int epicsTimeToTime_t (time_t *pDest, const epicsTimeStamp *pSrc);
int epicsTimeFromTime_t (epicsTimeStamp *pDest, time_t src);

/*convert to and from ANSI C's "struct tm" with nano seconds */
int epicsTimeToTM (struct tm *pDest, unsigned long *pNSecDest,

```

```
    const epicsTimeStamp *pSrc);
int epicsTimeToGMTM (struct tm *pDest, unsigned long *pNSecDest,
    const epicsTimeStamp *pSrc);
int epicsTimeFromTM (epicsTimeStamp *pDest, const struct tm *pSrc,
    unsigned long nSecSrc);

/* convert to and from POSIX RT's "struct timespec" */
int epicsTimeToTimespec (struct timespec *pDest, const epicsTimeStamp *pSrc);
int epicsTimeFromTimespec (epicsTimeStamp *pDest, const struct timespec *pSrc);

/* convert to and from BSD's "struct timeval" */
int epicsTimeToTimeval (struct timeval *pDest, const epicsTimeStamp *pSrc);
int epicsTimeFromTimeval (epicsTimeStamp *pDest, const struct timeval *pSrc);
/*arithmetic operations */
double epicsTimeDiffInSeconds (
    const epicsTimeStamp *pLeft, const epicsTimeStamp *pRight);
void epicsTimeAddSeconds (
    epicsTimeStamp *pDest, double secondsToAdd); /* adds seconds to *pDest */

/*comparison operations: returns (0,1) if (false,true) */
int epicsTimeEqual(const epicsTimeStamp *pLeft, const epicsTimeStamp *pRight);
int epicsTimeNotEqual(const epicsTimeStamp *pLeft,const epicsTimeStamp *pRight);
int epicsTimeLessThan(const epicsTimeStamp *pLeft,const epicsTimeStamp *pRight);
int epicsTimeLessThanEqual(
    const epicsTimeStamp *pLeft, const epicsTimeStamp *pRight);
int epicsTimeGreaterThan (
    const epicsTimeStamp *pLeft, const epicsTimeStamp *pRight);
int epicsTimeGreaterThanEqual (
    const epicsTimeStamp *pLeft, const epicsTimeStamp *pRight);
/*convert to ASCII string */
size_t epicsTimeToStrftime (
    char *pBuff, size_t bufLength, const char *pFormat, const epicsTimeStamp
    *pTS);

/* dump current state to standard out */
void epicsTimeShow (const epicsTimeStamp *, unsigned interestLevel);
/* OS dependent reentrant versions of the ANSI C interface because */
/* vxWorks gmtime_r interface does not match POSIX standards */
int epicsTime_localtime ( const time_t *clock, struct tm *result );
int epicsTime_gmtime ( const time_t *clock, struct tm *result );
```

The C interface provides most of the features as the C++ interface. The features of the C++ operators are provided as functions.

20.14 *osiPoolStatus*

osiPoolStatus.h contains the following description:

```
int osiSufficientSpaceInPool(void);
```

Method	Meaning
osiSufficientSpaceInPool	Return (true,false) if there is sufficient free memory.

This determines if enough free memory exists to continue.

A vxWorks version returns (true,false) if memFindMax returns (>100000, <=100000) bytes.

The default version always returns true.

20.15 *osiProcess*

osiProcess.h contains the following:

```
typedef enum osiGetUserNameReturn {
    osiGetUserNameFail,
    osiGetUserNameSuccess
}osiGetUserNameReturn;
```

```
osiGetUserNameReturn osiGetUserName (char *pBuf, unsigned bufSize);
```

```
/*
 * Spawn detached process with named executable, but return
 * osiSpawnDetachedProcessNoSupport if the local OS does not
 * support heavy weight processes.
 */
```

```
typedef enum osiSpawnDetachedProcessReturn {
    osiSpawnDetachedProcessFail,
    osiSpawnDetachedProcessSuccess,
    osiSpawnDetachedProcessNoSupport
}osiSpawnDetachedProcessReturn;
```

```
osiSpawnDetachedProcessReturn osiSpawnDetachedProcess(
    const char *pProcessName, const char *pBaseExecutableName);
```

Not documented.

20.16 *osiSigPipeIgnore*

osiSigPipeIgnore.h contains the following:

```
void installSigPipeIgnore (void);
```

Not documented.

20.17 *osiSock.h*

See the header file in <base>/src/libCom/osi.

20.18 *Device Support Library*

NOTE: EPICS Base only provides vxWorks and RTEMS back-end implementations of these routines. Versions of the back-end routines for other operating systems can be added in a support or IOC application.

20.18.1 Overview

devLib.h provides definitions for a library of routines useful for device and driver modules, which are primarily intended for accessing VME devices. If all VME drivers register with these routines then addressing conflicts caused by multiple device/drivers trying to use the same VME addresses will be detected.

20.18.2 Location Probing

20.18.2.1 Read Probe

```
long devReadProbe(
    unsigned wordSize,
    volatile const void *ptr,
    void *pValueRead);
```

Performs a bus-error-safe atomic read operation width `wordSize` bytes from the `ptr` location, placing the value read (if successful) at `pValueRead`. The routine returns a failure status (non-zero) if a bus error occurred during the read cycle.

20.18.2.2 Write Probe

```
long devWriteProbe(
    unsigned wordSize,
    volatile void *ptr,
    const void *pValueWritten);
```

Performs a bus-error-safe atomic write operation width `wordSize` which copies the value from `pValueWritten` to the `ptr` location. The routine returns a failure status (non-zero) if a bus error occurred during the write cycle.

20.18.2.3 No Response Probe

```
long devNoResponseProbe(
    epicsAddressType addrType,
    size_t base,
    size_t size);
```

This routine performs a series of read probes for all word sizes from char to long at every naturally aligned location in the range `[base, base+size)` for the given bus address type. It returns an error if any location responds or if any such location cannot be mapped.

20.18.3 Registering VME Addresses

20.18.3.1 Definitions of Address Types

```
typedef enum {
    atVMEA16, atVMEA24, atVMEA32,
    atISA,
    atLast /* atLast is the last enum in this list */
} epicsAddressType;

char *epicsAddressTypeName[] = {
    "VME A16", "VME A24", "VME A32",
    "ISA"
};
```

20.18.3.2 Register Address

```
long devRegisterAddress(
    const char *pOwnerName,
    epicsAddressType addrType,
    size_t logicalBaseAddress,
    size_t size, /* bytes */
    volatile void **pLocalAddress);
```

This routine is called to register a VME address. The routine keeps a list of all VME address ranges requested and returns an error message if an attempt is made to register any addresses that overlap a range that is already being used. *pLocalAddress is set equal to the address as seen by the caller.

20.18.3.3 Print Address Map

```
long devAddressMap(void)
```

This routine displays the table of registered VME address ranges, including the owner of each registered address.

20.18.3.4 Unregister Address

```
long devUnregisterAddress(
    epicsAddressType addrType,
    size_t logicalBaseAddress,
    const char *pOwnerName);
```

This routine releases address ranges previously registered by a call to devRegisterAddress or devAllocAddress.

20.18.3.5 Allocate Address

```
long devAllocAddress(
    const char *pOwnerName,
    epicsAddressType addrType,
    size_t size,
    unsigned alignment, /*number of low zero bits needed in addr*/
    volatile void **pLocalAddress);
```

This routine is called to request the library to allocate an address block of a particular address type. This is useful for devices that appear in more than one address space and can program the base address of one window using registers found in another window.

20.18.4 Interrupt Connection Routines

20.18.4.1 Connect

```
long devConnectInterruptVME(  
    unsigned vectorNumber,  
    void (*pFunction)(void *),  
    void *parameter);
```

Connect ISR pFunction up to the VME interrupt vectorNumber.

20.18.4.2 Disconnect

```
long devDisconnectInterruptVME(  
    unsigned vectorNumber,  
    void (*pFunction)(void *));
```

Disconnects an ISR from its VME interrupt vector. The parameter pFunction should be set to the C function pointer that was connected. It is used as a key to prevent a driver from inadvertently removing an interrupt handler that it didn't install.

20.18.4.3 Check If Used

```
int devInterruptInUseVME(  
    unsigned vectorNumber);
```

Determines if a VME interrupt vector is in use, returning a boolean value.

20.18.4.4 Enable

```
long devEnableInterruptLevelVME(  
    unsigned level);
```

Enable the given VME interrupt level onto the CPU.

20.18.4.5 Disable

```
long devDisableInterruptLevelVME(  
    unsigned level);
```

Disable VME interrupt level. This routine should generally never be used, since it is impossible for a driver to know whether any other active drivers are still making use of a particular interrupt level.

20.18.5 Macros for Normalized Analog Values

20.18.5.1 Convert Digital Value to a Normalized Double Value

```
#define devCreateMask(NBITS)((1<<(NBITS))-1)  
#define devDigToNml(DIGITAL,NBITS) \  
    (((double)(DIGITAL))/devCreateMask(NBITS))
```

20.18.5.2 Convert Normalized Double Value to a Digital Value

```
#define devNmlToDig(NORMAL,NBITS) \  
    (((long)(NORMAL)) * devCreateMask(NBITS))
```

20.18.6 Deprecated Interrupt Routines

20.18.6.1 Definitions of Interrupt Types (deprecated)

```
typedef enum {intCPU, intVME, intVXI} epicsInterruptType;
```

The routines that use this typedef have all been deprecated, and currently only exist for backwards compatibility purposes. The typedef will be removed in a future release, along with those routines.

20.18.6.2 Connect (deprecated)

```
long devConnectInterrupt(  
    epicsInterruptType intType,  
    unsigned vectorNumber,  
    void (*pFunction)(),  
    void *parameter);
```

This routine has been deprecated, and currently only exists for backwards compatibility purposes. Uses of this routine should be converted to call `devConnectInterruptVME` or related routines instead. This routine will be removed in a future release.

20.18.6.3 Disconnect (deprecated)

```
long devDisconnectInterrupt(  
    epicsInterruptType intType,  
    unsigned vectorNumber);
```

This routine has been deprecated, and currently only exists for backwards compatibility purposes. Uses of this routine should be converted to call `devDisconnectInterruptVME` or related routines instead. This routine will be removed in a future release.

20.18.6.4 Enable Level (deprecated)

```
long devEnableInterruptLevel(  
    epicsInterruptType intType,  
    unsigned level);
```

This routine has been deprecated, and currently only exists for backwards compatibility purposes. Uses of this routine should be converted to call `devEnableInterruptLevelVME` or related routines instead. This routine will be removed in a future release.

20.18.6.5 Disable Level (deprecated)

```
long devDisableInterruptLevel(  
    epicsInterruptType intType,  
    unsigned level);
```

This routine has been deprecated, and currently only exists for backwards compatibility purposes. Uses of this routine should be converted to call `devDisableInterruptLevelVME` or related routines instead. This routine will be removed in a future release.

20.19 vxWorks Specific routines

The routines described in this section are included in a application by the Makfile command:

```
<appl>_OBJS_vxWorks += $(EPICS_BASE_BIN)/vxComLibrary
```

20.19.1 iocClock

This provides a clock for vxWorks by using Network Time Protocol (NTP) client calls. It is required for all vxWorks IOCs. It also provides the `date` command.

20.19.2 veclist

For VME systems this shows the interrupt vectors.

20.19.3 logMsgToErrlog

This traps all calls to `logMsg` and sends them to `errlogPrintf`.

20.19.4 drvTS

This is a driver that supports hardware event systems. It is documented in `TSdriver.html`, which is in the `documentation` directory of `base`. Unless `TSinit` is called before `iocInit` it will not be used.

20.19.5 camacLib.h

This was included with 3.13.

20.19.6 epicsDynLink

This provides `symFindByNameEPICS`. It is only provided for device/driver support that has not been converted to use `epicsFindSymbol`. Some day `epicsDynLink` will no longer be supported.

20.19.7 module_types.h

It is only provided for device/driver support that has not been converted to use OSI features of `base`. Some day it will no longer be supported. Instead of using this drivers should accept a `configure` command that specifies the information provided by `module_types.h`

20.19.8 task_params.h

It is only provided for device/driver support that has not been converted to use OSI features of `base`. Some day it will no longer be supported.

20.19.9 vxComLibrary

This is a routine that causes `iocClock`, `drvTS`, `epicsDynLink`, `logMsgToErrlog` and `veclist` to be loaded.

Chapter 21: Registry

Under vxWorks `osiFindGlobalSymbol` can be used to dynamically bind to record, device, and driver support and functions for use with subroutine records. Since on some systems this always returns failure, a registry facility is provided to implement the binding. The basic idea is that any storage meant to be "globally" accessible must be registered before it can be accessed by other code.

A perl script is provided that reads the `xxxApp.dbd` file and produces a c file containing a routine `registerRecordDeviceDriver`, which registers all record/device/driver/function support defined in the `xxxApp.dbd` file.

21.1 Registry.h

```
int registryAdd(void *registryID,const char *name,void *data);
void *registryFind(void *registryID,const char *name);
int registrySetTableSize(int size);
void registryFree();
int registryDump(void);
```

This is the code which does the work. Each different set of things to register must have its own unique ID. Everything to be registered is stored in the same gpHash table.

Routine `registrySetTableSize` is provided in case the default hash table size (1024 entries) is not sufficient.

21.2 registryRecordType.h

```
typedef int (*computeSizeOffset)(dbRecordType *pdbRecordType);

typedef struct recordTypeLocation {
    struct rset *prset;
    computeSizeOffset sizeOffset;
}recordTypeLocation;

int registryRecordTypeAdd(const char *name,recordTypeLocation *prtl);
recordTypeLocation *registryRecordTypeFind(const char *name);
```

Some features:

- Access to both the record support entry table and to the routine which computes the size and offset of each field are provided
- Type safe access is provided.

21.3 registryDeviceSupport.h

```
int registryDeviceSupportAdd(const char *name,struct dset *pdset)
struct dset *registryDeviceSupportFind(const char *name);
```

This provides access to the device support entry table.

21.4 registryDriverSupport.h

```
int registryDriverSupportAdd(const char *name,struct drvet *pdrvet);
struct drvet *registryDriverSupportFind(const char *name);

/* The following function is generated by registerRecordDeviceDriver/pl */
int registerRecordDeviceDriver(DBBASE *pdbbase);
```

This provides access to the driver support table.

21.5 registryFunction.h

```
typedef void (*REGISTRYFUNCTION)(void);

typedef struct registryFunctionRef {
    const char *name;
    REGISTRYFUNCTION addr;
}registryFunctionRef;

/* c interface definitions */
int registryFunctionAdd(const char *name,REGISTRYFUNCTION func);
REGISTRYFUNCTION registryFunctionFind(const char *name);
int registryFunctionRefAdd(
    registryFunctionRef ref[],int nfunctions);
```

registryFunctionAdd registers a single function. registryFunctionRefAdd registers several functions.

For example subroutines for subRecord could be registered as follows:

```
#include <stdio.h>
#include "subRecord.h"

typedef long (*processMethod)(subRecord *precord);

static long mySubInit(subRecord *precord, processMethod process)
{
    printf("%s mySubInit process %p\n",precord->name, (void*) process);
    return(0);
}

static long mySubProcess(subRecord *precord)
{
    printf("%s mySubProcess\n",precord->name);
```

```

    return(0);
}
static registryFunctionRef mySubRef[] = {
    {"mySubInit", (REGISTRYFUNCTION)mySubInit},
    {"mySubProcess", (REGISTRYFUNCTION)mySubProcess}
};

void mySub(void)
{
    registryFunctionRefAdd(mySubRef, NELEMENTS(mySubRef));
}
epicsExportRegistrar(mySub);

```

The example registers these functions at application startup using an entry in the dbd file, which are converted by the registerRecordDeviceDriver.pl script into the necessary registration C source code. The dbd file entry is:

```
registrar("mySub")
```

The functions could also be registered using the epicsRegisterFunction macro in the C source file:

```
epicsRegisterFunction(mySubInit);
epicsRegisterFunction(mySubProcess);
```

and the function keyword in the application database definition:

```
function(mySubInit)
function(mySubProcess)
```

21.6 registerRecordDeviceDriver.c

A version of this is provided for vxWorks. This version makes it unnecessary to use registerRecordDeviceDriver.pl or register other external names. Thus for vxWorks everything can work almost exactly like it did in release 3.13.x

21.7 registerRecordDeviceDriver.pl

This is the perl script which creates a c source file that registers record/device/driver/function support. Make rules:

- execute this script using the dbd file created by dbExpand
- compile the resulting C file
- Make the object file part of the xxxLib file

Chapter 22: Database Structures

22.1 Overview

This chapter describes the internal structures describing an IOC database. It is of interest to EPICS system developers but serious application developers may also find it useful. This chapter is intended to make it easier to understand the IOC source listings. It also gives a list of the header files used by IOC Code.

22.2 Include Files

This section lists the files in base/include that are of most interest to IOC Application Developers:

alarm.h alarmString.h - These files contain definitions for all alarm status and severity values.

caDef.h caErr.h caEventMask.h - These files are of interest to anyone writing channel access clients.

callback.h - The definitions for the General Purpose callback system.

dbAccess.h - Definitions for the runtime database access routines.

dbBase.h - Definitions for the structures used to store an EPICS database.

dbDefs.h - A catchall file for definitions that have no other reasonable place to appear.

dbFldTypes.h - Definitions for DBF_XXXX and DBR_XXXX types.

dbScan.h - Definitions for the scanning system.

dbStaticLib.h - The static databases access system.

db_access.h db_addr.h - Old database access.

devLib.h - The device support library

devSup.h - Device Support Modules

drvSup.h - Driver Support Modules

ellLib.h - A library that provides the same functions as the vxWorks `lstLib`. All routines start with `ell` instead of `lst`. The `ellLib` routines work on both IOCs and on UNIX.

epicsPrint.h errMdef.h - EPICS error handling system

fast_lock.h - The FASTLOCK routines.

freeList.h - A general purpose free list facility

gpHash.h - A general purpose hash library.

guigroup.h - The guigroup definitions.

initHooks.h - Definitions used by `initHooks.c` routines.

Chapter 22: Database Structures

Include Files

link.h - Link definitions

module_types.h - VME hardware configuration. **SHOULD NOT BE USED BY NEW SUPPORT.**

recSup.h - The record global routines.

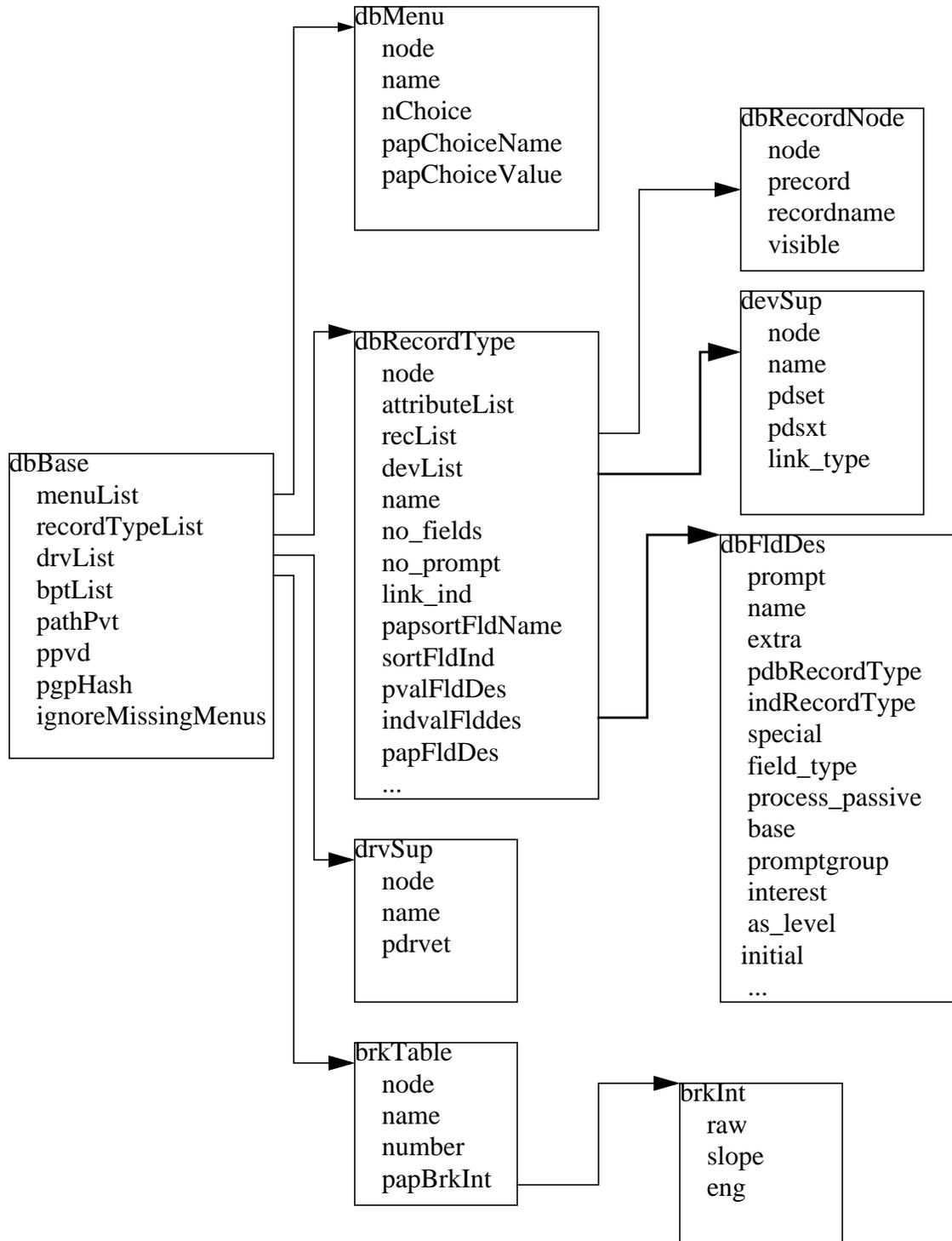
special.h - Definitions for special fields, i.e. `SPC_XXX`.

task_params.h - Definitions for task priorities, stack space, etc.

taskwd.h - Task Watchdog System

tsDefs.h - Time stamp routines. Will also have to look at `base/src/libCom/tsSubr.c`

22.3 Structures



Chapter 23: INDEX

A	
AB_IO - link field value	105
Access Security	125
Configuration File	126
Initialization	128
Access Security Group	129
Access Security Level	138
add_record	187
addpath	92, 94
adjustment.h	273
adjustToWorstCaseAlignment	273
algorithm	258
Alloc/Free DBENTRY	195
allocate	265
ANSI	69
archclean	40
asAddClient	135
asAddMember	134
ascar	153
asChangeClient	135
asChangeGroup	135
ascheck	128
asCheckGet	136
asCheckPut	136
asCompute	137
asComputeAllAsg	136
asComputeAsg	137
asdbdump	140, 151
asdbdumpFP	140
asDbGetAsl	139
asDbGetMemberPvt	139
asDump	137
asDumpFP	137
asDumpHag	137
asDumpHagFP	137
asDumpHash	138
asDumpHashFP	138
asDumpMem	138
asDumpMemFP	138
asDumpRules	137
asDumpRulesFP	137
asDumpUag	137
asDumpUagFP	137
ASG	126
ASG - access security configuration	127
asGetClientPvt	136
asGetMemberPvt	135
asInit	120, 129, 139, 151
asInitAsyn	139
asInitFile	134
asInitFP	134
asInitialize	134
ASL	126
asl - field definition rules	95
asl_level - field definition	96
ASL0	138
ASL1	138
asphag	140, 152
asphagFP	140
aspmem	140, 152
aspmemFP	140
asprules	140, 152
asprulesFP	140
aspuag	140, 151
aspuagFP	140
asPutClientPvt	136
asPutMemberPvt	135
asPvt in DBADDR	211
asRegisterClientCallback	136
asRemoveClient	136
asRemoveMember	134
asSetFilename	128, 139, 151
asSetSubstitutions	129, 139
asSubInit	129, 140
asSubProcess	129, 140
astac	139
asTrapWriteAfter	136, 142
asTrapWriteBefore	136, 142
asTrapWriteId	142
asTrapWriteListener	142
asTrapWriteMessage	142
asTrapWriteRegisterListener	142
asTrapWriteUnregisterListener	142
asynchronous device support example	183
aToIPAddr	272
B	
base - field definition	98
base - field definition rules	96
base.dbd	42
BBGPB_IO - link field value	105
BIN_INSTALLS	59, 72
BITBUS_IO - link field value	105
borland.bat	77
breakpoint table - database definition	103
Breakpoint Tables	41, 107
Breakpoints	147
breaktable	93
bucketLib.h	253
Build Facility	31
Build software prerequisites	33
BuildingR3.13AppsWithR3.14.html	76
BuildingR3.13ExtensionsWithR3.14.html	76
C	
C++ library	257
ca_put_callback	87
Cached Puts	87
CALC - access security configuration	128
CALC_ERR_	253

calcArgUsage	253
calcErrorStr	253
calcPerform	253
CALLBACK	229
callbackGetUser	230
callbackInit	230
callbackRequest	230
callbackRequestDelayed	230
callbackRequestProcessCallback	230
callbackRequestProcessCallbackDelayed	230
callbackSetCallback	230
callbackSetPriority	230
callbackSetQueueSize	120, 231–232
callbackSetUser	230
calloc	273
callocMustSucceed	273
CAMAC_IO - link field value	105
camacLib.h	305
cancel	264
cantProceed	273
cantProceed.h	273
casr	152
Cexp	22
Channel Access	23
channel access link	79
Channel Access Monitors	223
checkAlarms	171
choice	92
choice_string - device definition	100
class templates	257
CLASSES	71
clean	39
CMPLR	69
comment - Database Definitions	94
Compiler options	44
computeSizeOffset	307
CONFIG	73
CONFIG.CrossCommon	73
CONFIG.gnuCommon	73
CONFIG_ADDONS	73
CONFIG_APP_INCLUDE	35
CONFIG_BASE	73
CONFIG_BASE_VERSION	73
CONFIG_COMMON	73
CONFIG_ENV	73
CONFIG_SITE	34, 73
CONFIG_SITE.Common.target	34
CONFIG_SITE.Common.vxWorksCommon	34
CONFIG_SITE.host.Common	34
CONFIG_SITE.host.host	34
CONFIG_SITE.host.target	34
CONFIG_SITE_ENV	34, 73
Configurating base	34
Configure files	73
configure/os File	74
configure/tool File	74
CONSTANT - link field value	104
constant link	79
ConvertingR3.13AppsToR3.14.html	76
convertRelease.pl	74–75
coreRelease	154
cp.pl	75
createTimer	264
cross target	32

CROSS_COMPILER_HOST_ARCHS	32
CROSS_COMPILER_TARGET_ARCHS	32, 35–36
CROSS_OPT	69
CROSS_WARN	69
cvscan	40, 75
cvt_dbaddr - Record Support Routine	174
cvtBitsToUlong	258
cvtCharToString	258
cvtDoubleToCompactString	258
cvtDoubleToExpString	258
cvtDoubleToString	258
cvtFast.h	258
cvtFloatToCompactString	258
cvtFloatToExpString	258
cvtFloatToString	258
cvtLongToHexString	258
cvtLongToOctalString	258
cvtLongToString	258
cvtShortToString	258
cvtUcharToString	258
cvtUlongToBits	258
cvtUlongToString	258
cvtUshortToString	258
CXXCMPLR	69

D

database access routines - List of	212
Database Definition	42–43
Database Definition File	91
database definitions	91
Database Files	43
Database Format - Summary	91
database link	79
Database Link Guidelines	82
Database Links	79
Database Locking	80
Database Scanning	81
DB	44, 71
DB install files	44
DB_INSTALLS	44, 71
DB_MAX_CHOICES	209
db_post_events	223
dba	146
dbAccess.h	209
dbAdd	210
dbAddPath	196
DBADDR	211
dbAllocBase	194
dbAllocEntry	195
dbAllocForm	204
dbap	148
dbb	148
dbBufferSize	219
dbc	148
dbCaAddLink	225
dbCaAddLinkCallback	225
dbCaGetAlarmLimits	228
dbCaGetAttributes	227
dbCaGetControlLimits	228
dbCaGetGraphicLimits	228
dbCaGetLink	226

dbCaGetPrecision	228	dbFirstField	198
dbCaGetSevr	227	dbFirstInfo	205
dbCaGetTimeStamp	227	dbFirstRecord	200
dbCaGetUnits	228	dbFirstRecordType	197
dbCaLinkInit	119, 225	dbFldTypes.h	209
dbCaPutLink	226	dbFoundField	198, 201
dbCaPutLinkCallback	226	dbFreeBase	195
dbcar	153, 155	dbFreeEntry	195
dbCaRemoveLink	226	dbFreeForm	204
dbCopyEntry	196	dbGet	215
dbCopyEntryContents	196	dbGetDefaultName	199
dbCopyRecord	200–201	dbGetField	215
dbCreateRecord	200	dbGetFieldIndex	220
dbCvtLinkToConstant	203	dbGetFieldName	198
dbCvtLinkToPvlink	203	dbGetFieldType	198
DBD	41–43, 71	dbGetFormPrompt	204
dbd.	148	dbGetFormValue	204
DBD install files	43	dbGetInfo	206
DBD_INSTALLS	43, 71	dbGetInfoName	205
dbDefs.h	209, 273	dbGetInfoPointer	205
dbDeleteInfo	206	dbGetInfoString	205
dbDeleteRecord	200	dbGetLink	215
DBDINC	41–42, 71	dbGetLinkDBFtype	221
dbDumpBreaktable	206	dbGetLinkField	203
dbDumpDevice	157, 206	dbGetLinkType	203
dbDumpDriver	158, 206	dbGetMenuChoices	202
dbDumpField	157	dbGetMenuIndex	202
dbDumpFldDes	206	dbGetMenuIndexFromString	202
dbDumpFunction	206	dbGetMenuStringFromIndex	202
dbDumpMenu	157, 206	dbGetNelements	220
dbDumpPath	206	dbGetNFields	198
dbDumpRecord	206	dbGetNLinks	203
dbDumpRecords	158, 207	dbGetNMenuChoices	202
dbDumpRecordType	157, 206	dbGetNRecords	200
dbDumpRegistrar	206	dbGetNRecordTypes	197
dbDumpVariable	206	dbGetPdbAddrFromLink	220
DBE_ALARM	173	dbGetPrompt	199
DBE_LOG	173	dbGetPromptGroup	199
DBE_VAL	173	dbGetRange	201
dbel	153	dbGetRecordAttribute	199
dbExpand	42, 112, 207	dbGetRecordName	200
DBF_CHAR	209	dbGetRecordTypeName	198
DBF_DEVICE	210	dbGetRset	220
DBF_DOUBLE	210	dbGetString	201
DBF_ENUM	210	dbgf	146
DBF_FLOAT	210	dbgrep	146
DBF_FWDLINK	210	dbhcr	150, 156
DBF_INLINK	210	dbInitEntry	195
DBF_LONG	210	dbInvisibleRecord	201
DBF_MENU	210	dbior	149
DBF_NOACCESS	210	dbIsDefaultValue	201
DBF_OUTLINK	210	dbIsLinkConnected	220
DBF_SHORT	210	dbIsValueField	220
DBF_STRING	209	dbIsVisibleRecord	201
DBF_UCHAR	210	dbl	145
DBF_ULONG	210	dbLoadDatabase	112
DBF_USHORT	210	dbLoadRecords	113
dbFindBrkTable	206	dbLoadTemplate	113
dbFindField	201	dbLockGetLockId	224
dbFindInfo	205	dbLockInitRecords	119, 224
dbFindMenu	202	dbLockSetGblLock	224
dbFindRecord	200	dbLockSetGblUnlock	224
dbFindRecordType	197	dbLockSetMerge	224
dbFinishEntry	195	dbLockSetRecordLock	225

dbLockSetSplitSI	224	dbRenameRecord	201
dblsr	155	dbReportDeviceConfig	206
dbmf.h	259	dfs	148
dbmfFree	259	dbScan.h	237
dbmfFreeChunks	259	dbScanFwdLink	222
dbmfInit	259	dbScanLink	222
dbmfMalloc	259	dbScanLock	224
dbmfShow	259	dbScanPassive	222
dbNameToAddr	214	dbScanUnlock	224
dbNextField	198	dbstat	149
dbNextInfo	205	dbtgf	154
dbNextRecord	200	dbToMenuH	109
dbNextRecordType	197	dbToRecordtypeH	109–110
dbNotifyAdd	219	dbtptf	154
dbNotifyCancel	219	dbtpn	155
dbNotifyCompletion	219	dbtr	147
dbnr	147	dbTranslateEscape	93, 273
dbp	148	dbValueSize	219
dbPath	196	dbVerify	201
dbpf	146	dbVerifyForm	204
dbpr	147	dbVisibleRecord	201
dbProcess	222	dbWriteBreaktable	196
dbPut	217	dbWriteBreaktableFP	196
dbPutAttribute	107, 222	dbWriteDevice	196
dbPutField	216	dbWriteDeviceFP	196
dbPutForm	204	dbWriteDriver	196
dbPutInfo	206	dbWriteDriverFP	196
dbPutInfoPointer	205	dbWriteFunctionFP	196
dbPutInfoString	205	dbWriteMenu	196
dbPutLink	216	dbWriteMenuFP	196
dbPutMenuIndex	202	dbWriteRecord	197
dbPutNotify	87, 217–218	dbWriteRecordFP	197
dbPutNotifyInit	120	dbWriteRecordType	196
dbPutRecordAttribute	199	dbWriteRecordTypeFP	196
dbPutString	201	dbWriteRegistrarFP	196
dbPvdDump	158, 206	dbWriteVariableFP	196
dbPvdTableSize	120	DCT_FWDLINK	194
DBR_AL_DOUBLE	213	DCT_INLINK	194
DBR_AL_LONG	213	DCT_INTEGER	194
DBR_CHAR	213	DCT_LINK_CONSTANT	203
DBR_CTRL_DOUBLE	213	DCT_LINK_DEVICE	203
DBR_CTRL_LONG	213	DCT_LINK_FORM	203
DBR_DOUBLE	213	DCT_LINK_PV	203
DBR_ENUM	213	DCT_MENU	194
DBR_ENUM_STRS	213	DCT_MENUFORM	194
dbr_field_type in DBADDR	211	DCT_NOACCESS	194
DBR_FLOAT	213	DCT_OUTLINK	194
DBR_GR_DOUBLE	213	DCT_REAL	194
DBR_GR_LONG	213	DCT_STRING	194
DBR_LONG	213	del_record	188
DBR_PRECISION	213	destroy	264
DBR_PUT_ACKS	213–214	devAddressMap	302
DBR_PUT_ACKT	213–214	devAllocAddress	302
DBR_SHORT	213	devConnectInterrupt	304
DBR_STATUS	213	devConnectInterruptVME	303
DBR_STRING	213	devCreateMask	303
DBR_TIME	213	devDisableInterruptLevel	304
DBR_UCHAR	213	devDisableInterruptLevelVME	303
DBR_ULONG	213	devDisconnectInterrupt	304
DBR_UNITS	213	devDisconnectInterruptVME	303
DBR_USHORT	213	devEnableInterruptLevel	304
dbReadDatabase	196	devEnableInterruptLevelVME	303
dbReadDatabaseFP	196	device	92
dbReadTest	115	device - database definition	100

Device Support Entry Table	167	EPICS_HOST_ARCH	33
devInterruptInUseVME	303	EPICS_IOC_LOG_FILE_COMMAND	163
devNmIToDig	303	EPICS_IOC_LOG_FILE_LIMIT	163
devNoResponseProbe	301	EPICS_IOC_LOG_FILE_NAME	163
devReadProbe	301	EPICS_IOC_LOG_INET	123
devRegisterAddress	302	EPICS_IOC_LOG_PORT	123, 164
devUnregisterAddress	302	EPICS_THREAD_ONCE_INIT	287
devWriteProbe	301	EPICS_TS_NTP_INET	123
Directory structure	31	epicsAddressType	302
distclean	39	epicsAddressTypeName	302
DLL_LIBS	50	epicsAlgorithm.h	258
Doc file	53	epicsAssert	278
DOCS	53, 71	epicsAtExit	257
Documentation Files	75	epicsAtThreadExit	257
driver	92	epicsDynLink	305
driver - database definition	101	epicsEnvSet	123
Driver Support Entry Table Example	190	epicsEnvShow	123, 154
drvnet_name - driver definition	101	epicsEvent	278
drvTS	305	epicsEvent.h	278
DSET	167	epicsEventCreate	279
dset - dbCommon	181	epicsEventDestroy	279
dset_name - device definition	100	epicsEventId	279
dtyp - dbCommon	181	epicsEventMustCreate	279
		epicsEventMustWait	280
		epicsEventShow	280
		epicsEventSignal	280
		epicsEventTryWait	280
		epicsEventWait	280
		epicsEventWaitError	278
		epicsEventWaitOK	278
		epicsEventWaitStatus	278
		epicsEventWaitTimeout	278
		epicsEventWaitWithTimeout	280
		epicsExit	257
		epicsExitCallAtExits	257
		epicsExitCallAtThreadExits	257
		epicsExport	15
		epicsExportAddress	15, 102
		epicsExportRegistrar	101, 251
		epicsFindSymbol	280
		epicsFindSymbol.h	280
		epicsGetStderr	285
		epicsGetStdin	285
		epicsGetStdout	285
		epicsGetThreadStderr	285
		epicsGetThreadStdin	285
		epicsGetThreadStdout	285
		EpicsHostArch	76
		EpicsHostArch.pl	33, 76
		epicsInterrupt	280
		epicsInterrupt.h	280
		epicsInterruptContextMessage	280
		epicsInterruptIsInterruptContext	280
		epicsInterruptLock	280
		epicsInterruptType	304
		epicsInterruptUnlock	280
		epicsMath	281
		epicsMath.h	281
		epicsMax	258
		epicsMessageQueue	281–283
		epicsMessageQueueCreate	283
		epicsMessageQueueDestroy	283
		epicsMessageQueuePending	283
		epicsMessageQueueReceive	283

E

E2DB_FLAGS	71
ellAdd	260
ellConcat	260
ellCount	260
ellDelete	260
ellExtract	260
ellFind	260
ellFirst	260
ellFree	260
ellGet	260
ellInit	260
ellInsert	260
ellLast	260
ellLib.h	260
ELLLIST	260
ellNext	260
ELLNODE	260
ellNStep	260
ellNth	260
ellPrevious	260
ellVerify	260
eltc	149, 162
eng_value - breakpoint table	103
Environment Prerequisites	33
Environment Variables	122
EPICS	7, 23
Basic Attributes	23
Hardware/Software Platforms	24
Overview	7
EPICS_CA_ADDR_LIST	122
EPICS_CA_AUTO_ADDR_LIST	122
EPICS_CA_BEACON_PERIOD	122
EPICS_CA_CONN_TMO	122
EPICS_CA_MAX_ARRAY_BYTES	122
EPICS_CA_REPEATER_PORT	122
EPICS_CA_SERVER_PORT	122

epicsMessageQueueReceiveWithTimeout	283	epicsTempName	285
epicsMessageQueueSend	283	epicsThread	286, 290
epicsMessageQueueSendWithTimeout	283	epicsThread.h	286
epicsMessageQueueShow	283	epicsThreadBooleanStatus	287
epicsMessageQueueTryReceive	283	epicsThreadBooleanStatusFail	287
epicsMessageQueueTrySend	283	epicsThreadBooleanStatusSuccess	287
epicsMin	258	epicsThreadCreate	287–288
epicsMutex	283	epicsThreadExitMain	287–288
epicsMutex.h	283	EPICSTHREADFUNC	286
epicsMutexCreate	284	epicsThreadGetId	287, 289
epicsMutexDestroy	284	epicsThreadGetIdSelf	287, 289
epicsMutexId	284	epicsThreadGetName	287, 289
epicsMutexLock	284	epicsThreadGetNameSelf	287, 289
epicsMutexLockError	283	epicsThreadGetPriority	287–288
epicsMutexLockOK	283	epicsThreadGetPrioritySelf	287–288
epicsMutexLockStatus	283	epicsThreadGetStackSize	287–288
epicsMutexLockTimeout	283	epicsThreadHighestPriorityLevelBelow	287–288
epicsMutexMustCreate	284	epicsThreadId	287
epicsMutexMustLock	284	epicsThreadInit	287
epicsMutexShow	284	epicsThreadIsEqual	287, 289
epicsMutexShowAll	284	epicsThreadIsOkToBlock	287, 289
epicsMutexTryLock	284	epicsThreadIsSuspended	287, 289
epicsMutexUnlock	284	epicsThreadLowestPriorityLevelAbove	287–288
epicsParamShow	154	epicsThreadOnce	287–288
epicsPrintf	161	epicsThreadOnceId	287
epicsPrtEnvParams	123	epicsThreadPriorityChannelAccessServer	287
epicsRegisterFunction	15, 102, 309	epicsThreadPriorityHigh	286
epicsRelease	154	epicsThreadPriorityLow	286
epicsRingBytes.h	260	epicsThreadPriorityMax	286
epicsRingBytesCreate	260	epicsThreadPriorityMedium	286
epicsRingBytesDelete	260	epicsThreadPriorityMin	286
epicsRingBytesFlush	260	epicsThreadPriorityScanHigh	287
epicsRingBytesFreeBytes	260	epicsThreadPriorityScanLow	287
epicsRingBytesGet	260	epicsThreadPrivateCreate	288–289
epicsRingBytesId	260	epicsThreadPrivateDelete	288–289
epicsRingBytesIsEmpty	260	epicsThreadPrivateGet	288, 290
epicsRingBytesIsFull	260	epicsThreadPrivateId	288
epicsRingBytesPut	260	epicsThreadPrivateSet	288–289
epicsRingBytesSize	260	epicsThreadResume	287–288
epicsRingBytesUsedBytes	260	epicsThreadRunnable	290
epicsRingPointer	261	epicsThreadSetOkToBlock	287, 289
epicsRingPointer.h	261, 281	epicsThreadSetPriority	287–288
epicsRtemsInitHooks.h	118	epicsThreadShow	288–289
epicsScanDouble	285	epicsThreadShowAll	288–289
epicsScanFloat	285	epicsThreadSleep	287, 289
epicsSetThreadStderr	285	epicsThreadSleepQuantum	287, 289
epicsSetThreadStdin	285	epicsThreadStackBig	287
epicsSetThreadStdout	285	epicsThreadStackMedium	287
epicsSprintf	285	epicsThreadStackSizeClass	287
epicsStdio	285	epicsThreadStackSmall	287
epicsStdio.h	285	epicsThreadSuspendSelf	287–288
epicsStdioRedirect	286	epicsTime	292–293, 295, 298
epicsStdlib	285	epicsTime.h	292
epicsStdlib.h	285	epicsTimeEvent	293, 295
epicsStdoutPrintf	285	epicsTimer	263–264
epicsStrCaseCmp	273	epicsTimer.h	263
epicsStrDup	274	epicsTimerId	266
epicsString.h	273	epicsTimerNotify	264
epicsStrnCaseCmp	273	epicsTimerQueueActive	265
epicsStrPrintEscaped	274	epicsTimerQueueId	266
epicsStrSnPrintEscaped	274	epicsTimerQueueNotify	265
epicsStrtod	285	epicsTimerQueuePassive	265
epicsSwap	258	epicsTimeStamp	292
epicsTempFile	285	epicsTypes.h	274

epicsVprintf 161
 epicsVsnprintf 285
 errlog Listeners 161
 errlogAddListener 161
 errlogFatal 160
 errlogFlush 160
 errlogGetSevEnumString 160
 errlogGetSevToLog 160
 errlogInfo 160
 errlogInit 120, 162
 errlogInit2 162
 errlogListener 161
 errlogMajor 160
 errlogMessage 159
 errlogMinor 160
 errlogPrintf 159, 178
 errlogRemoveListener 161
 errlogSetSevToLog 160
 errlogSevEnum 160
 errlogSevPrintf 160
 errlogSevVprintf 160
 errlogThread 162
 errlogVprintf 159
 errMessage 160
 errPrintf 160–161
 Escape Sequence 93
 Event 235
 Event - Scan Type 235
 Event Scanning 240
 EVNT - Scan Related Field 236
 exit 247
 expire 264
 Extended device support 181, 186
 extra - field definition rules 96
 extra_info - field definition 98

F

field 92
 field definition rules 95
 field value - record instance definition 104
 field_name - field definition 96
 field_name - record instance definition 104
 field_size in DBADDR 211
 field_type - field definition 96
 field_type in DBADDR 211
 filename extension conventions 94
 filterWarnings.pl 75
 freeList.h 269
 freeListCalloc 269
 freeListCleanup 269
 freeListFree 269
 freeListInitPvt 269
 freeListItemsAvail 269
 freeListMalloc 269
 fullpathname 75
 function 15, 309
 function templates 257
 function_name - function definition 102
 function_name - registrar definition 101
 FWDLINK 79

G

get_alarm_double Record Support Routine 177
 get_array_info - Record Support Routine 175
 get_control_double - Record Support Routine 177
 get_enum_str - record Support Routine 176
 get_enum_strs - record Support Routine 176
 get_graphic_double - example 170
 get_graphic_double - Record Support Routine 176
 get_ioint_info 239
 get_ioint_info - device support routine 185
 get_precision - Record Support Routine 175
 get_units - example 170
 get_units - Record Support Routine 175
 getExpireDelay 264
 getExpireInfo 264
 gft 156
 GNU make 33
 gnumake 39
 gphAdd 270
 gphHash.h 269
 gphDelete 270
 gphDump 270
 gphDumpFP 270
 GPENTRY 269
 gphFind 270
 gphFreeMem 270
 gphInitPvt 269
 GPIB_IO - link field value 105
 grecord 93
 gui_group - field definition 96
 Guidelines for Asynchronous Records 85
 Guidelines for Synchronous Records 84

H

HAG 126–128
 Header dependencies 40
 help 247
 Host makefile targets 37
 HOST_OPT 69
 HOST_WARN 69
 Html 53
 HTMLS 53, 71
 HTMLS_DIR 71

I

I/O Event - Scan Type 235
 I/O Event scanned 235
 I/O Event Scanning 238, 241
 INC 53, 70
 include 92
 include - Database Definitions 94
 Include File Generation 109
 Include files 53
 INFIX_TO_POSTFIX_SIZE 253
 info_name - record instance definition 106
 info_value - record instance definition 106

Information item pointer	107
Information Item	107
init - device support routine	185
init - Record Support Routine	174
init_record - device support routine	185
init_record - example	168
init_record - record support routine	174
init_value - field definition	96
InitDatabase	119
InitDevSup	119
InitDrvSup	119
initHookFunction	122
initHookRegister	122
initHooks	121
initHookState	122
initial - field definition rules	96
Initialize Logging	123
initialProcess	120
initPeriodic	242
InitRecSup	119
INLINK	79
INP - access security configuration	128
Input/Output Controller	7
Hardware/Software Platforms	24
Software Components	25
INPUTRC	248
INST_IO - link field value	105
Install Directories	31
Install Directory definitions	34
INSTALL_LOCATION	31, 34, 72
INSTALL_LOCATION_APP	31
INSTALL_LOCATION_EXTENSIONS	31
installEpics.pl	74-75
Installing Other Binaries	59
Installing Other Libraries	60
interest - field definition rules	96
interest_level - field definition	98
interruptAccept	120
IOC	23
See Input/Out Controller	
IOC Error Logging	159
Ioc makefile targets	37
iocClock	305
iocInit	118
iocLogClient	164
iocLogInit	123
iocLogServer	163
IOCSH_HISTSZ	248
IOCSH_PS1	248
iocshCmd	248
iocshSystemCommand	247
isinf	281
isnan	281
ISO C++	257

J

JAR	61, 71
JAR_INPUT	61, 72
JAR_MANIFEST	61, 72
JAVA	61
java C header file	71

Java class files	60
Java classes	71
Java Example	61
Java jar file	61
Java native methods	62
JAVAINC	62, 71

K

Keywords	92
KnownProblems.html	76

L

LAN	23
Lex and yac	54
LEXOPT	71
LIB_INSTALLS	60, 72
LIB_LDFLAGS	69
LIB_LIBS	49-50, 67
LIB_OBJLIBS	48, 66
LIB_OBJS	48, 65
LIB_RCS	60
LIB_SRCS	47, 65
LIB_SYS_LIBS	50, 67
LIBOBJ	49
Libraries	46
LIBRARY	63
Library example	50
Library link order	50
library name	46
Library object file	48
Library Source file	47
Library version number	50
LIBRARY_HOST	47, 63-64
LIBRARY_IOC	46, 64
LIBSRCS	47, 65
Linear Conversion	107
link options	44
link.h	209
LINK_ALARM	80
link_type - device definition	100
LINR	107
Loadable libraries	51
LOADABLE_LIBRARY	64
LOADABLE_LIBRARY	51
LOADABLE_LIBRARY_HOST	51, 64
Local Area Network	
Hardware/Software Platforms	24
logClientCreate	270
logClientFlush	270
logClientId	270
logClientInit	270
logClientSend	270
logClientSendMessage	271
logClientShow	270

M	
macCreateHandle	271
macDeleteHandle	272
macEnvExpand	272
macExpandString	271
macGetValue	271
macInstallMacros	272
macParseDefns	272
macPopScope	272
macPushScope	272
macPutValue	271
macReportMacros	272
Macro Substitution	93
macSuppressWarning	271
Make	39
Make commands	39
Make targets	40
makeConfigAppInclude.pl	74
Makefile	74
Makefiles	37–38
makeIncludeDbd	75
makeMakefile	75
makeMakefileInclude.pl	75
malloc	273
mallocMustSucceed	273
MANIFEST	72
MAX_STRING_SIZE	209
Maximize Severity	80
menu	92
menu - field definition rules	96
menuConvert	107
Menus	42
menuScan.dbd	236
mkdir.pl	75
mkmf.pl	75
module_types.h	305
monitor - example	172
MOTLOAD	21
MS	80
MSI	44
Multiple Definitions	94
munch.pl	75
mv.pl	75

N	
name - breakpoint table	103
name_CFLAGS	45, 68
name_CPPFLAGS	46, 68
name_CXXFLAGS	45, 68
name_DEPENDS	44
name_DIR	67
name_DLL_LIBS	50
name_INCLUDES	45, 69
name_LDFLAGS	46, 69
name_LDOBJS	49, 56, 66
name_LIBS	49–50, 57, 67
name_OBJLIBS	48, 55, 66
name_OBJS	48, 55, 66
name_RCS	60, 72

name_SNCFLAGS	71
name_SRCS	47, 52, 56, 65
name_SYS_LIBS	50, 57, 67
name_TEMPLATE	44
NMS	80
no_elements in DBADDR	211
NOTRAPWRITE	127
NPP	80

O	
Object Files	52
OBJLIB	51, 64
OBJLIB_OBJS	51, 64
OBJLIB_SRCS	51, 64
OBJS	52, 70
OBJS_HOST	52, 70
OBJS_IOC	52, 70
Operator Interface	
Hardware/Software Platforms	24
OPI	23
OS_CLASS	36
osclass	36
OSI_PATH_LIST_SEPARATOR	275
OSI_PATH_SEPARATOR	275
osiFindGlobalSymbol	280
osiMutex.h	299
osiPoolStatus.h	299
osiProcess.h	300
osiSem.h	300
osiSigPipeIgnore.h	300
osiSock.h	272, 301
osiSufficientSpaceInPool	299
OUTLINK	79
Overview of Record Processing	165

P	
Passive	235
Passive - Scan Type	235
path	92
path - Database Definitions	94
Path requirements	33
pdbbase	193
Periodic - Scan Type	235
Periodic Scanning	242
periodicTask	242
Perl	33
pfield in DBADDR	211
pfldDes in DBADDR	211
pft	156
PHAS - Scan Related Field	236
POSIX	41
Posix C source code	41
post_event	238, 240
postfix	253
postfix.h	253
PP	80
pp - field definition rules	96
pp_value - field definition	98

PPCBUG 21
 precord - DBADDR 211
 PRIO - Scan Related Field 236
 process - example 169
 process - Record Support Routine 174
 process - record support routine 82
 Process Passive 80
 PROD 54, 62
 PROD_HOST 54
 PROD_IOC 54, 62
 PROD_LDFLAGS 69
 PROD_LIBS 57, 67
 PROD_OBJLIBS 55, 66
 PROD_OBJS 55, 65
 PROD_RCS 60
 PROD_SRCS 56, 65
 PROD_SYS_LIBS 58, 67
 PROD_VERSION 58
 product libraries 49, 56
 Product link order 57
 product name 54
 product object file 55
 product source file 56
 product version number 58
 Products 54
 prompt - field definition rules 96
 prompt_value - field definition 96
 Psuedo field 107
 put_array_info - Record Support Routine 175
 put_enum_str - Record Support Routine 176
 putNotify 87
 putNotifyCanceled 219
 putNotifyCleanup 218
 putNotifyInit 218
 putNotifyOK 219
 PV_LINK - link field value 104
 PVNAME_SZ 209

Q

Quoted String 93

R

RANLIBFLAGS 71
 raw_value - breakpoint table 103
 RCS 60, 72
 README.1st 75
 README.darwin.html 76
 README.hpux 76
 README.html 75
 README.niCpu030 76
 README.WIN32 75
 realclean 39
 realuninstall 39
 rebuild 39
 recGblDbaddrError 178
 recGblFwdLink 179
 recGblGetAlarmDouble 179

recGblGetControlDouble 179
 recGblGetGraphicDouble 178
 recGblGetPrec 179
 recGblGetTimeStamp 179
 recGblInitConstantLink 179
 recGblRecordError 178
 recGblRecsupError 178
 recGblResetAlarms 177
 recGblSetSevr 177
 record 93
 record attribute 107
 record instance - database definition 104
 Record Instance File 91
 Record Processing 82
 Record Support Entry Table 166
 record type - Database Definition 95
 Record Type Definitions 41
 record_name - record instance definition 104
 record_type - device definition 100
 record_type - record instance definition 104
 record_type - record type definition 96
 recordtype 92
 Registering support routines 43
 registerRecordDeviceDriver 308
 registerRecordDeviceDriver.c 309
 registerRecordDeviceDriver.pl 309
 registrar 15, 93, 250
 iocsh commands 250
 registrar - database defintion 101
 Registry.h 307
 registryAdd 307
 registryDeviceSupport.h 308
 registryDeviceSupportAdd 308
 registryDeviceSupportFind 308
 registryDriverSupport.h 308
 registryDriverSupportAdd 308
 registryDriverSupportFind 308
 registryDump 307
 registryFind 307
 registryFree 307
 registryFunction.h 308
 registryFunctionAdd 308
 registryFunctionFind 308
 registryFunctionRef 308
 registryFunctionRefAdd 308
 registryRecordTypeAdd 307
 registrySetTableSize 307
 RELEASE 35–36, 73
 release 265
 RELEASE_DBDFLAGS 35
 RELEASE_INCLUDES 35
 RELEASE_NOTES.html 76
 ReleaseChecklist.html 76
 replaceVAR.pl 75
 report - device support routine 185
 report - Record Support Routine 173
 resource files 60
 resourceLib.h 259
 RF_IO - link field value 106
 ringPointer 261
 ringPointerCreate 262
 ringPointerDelete 263
 ringPointerFlush 263
 ringPointerGetFree 263

ringPointerGetSize	263	SPC_MOD	98
ringPointerGetUsed	263	SPC_NOMOD	97
ringPointerId	262	SPC_RESET	98
ringPointerIsEmpty	263	SPC_SCAN	97
ringPointerIsFull	263	special - field definition rules	96
ringPointerPop	263	special - Record Support Routine	174
ringPointerPush	263	special in DBADDR	211
rm.pl	75	special_value - field definition	97
RSET	166	Specifying libraries	56
RSET - example	167	Specifying libraries dependancies	49
RULE	128	spy	247
RULES	73	SRC_DIRS	41
RULES.Db	73	SRCS	47, 56, 64
RULES.ioc	74	standard C++ library	257
RULES_ARCHS	73	start	264
RULES_BUILD	73	Startup Files	76
RULES_DIRS	74	Startup files	33
RULES_INCLUDE	35	State Notation Programs	52
RULES_JAVA	74	STATIC_BUILD	69
RULES_TOP	74	status codes	162
<hr/> S <hr/>			
SCAN - Scan Related Field	235	std::max	258
Scan Once - Scan Type	235	std::min	258
Scan Related Database Fields	235	std::swap	258
SCAN_1ST_PERIODIC	237	STRICT	69
scanAdd	237	strtod	285
scanDelete	237	struct dbAddr	210
scanInit	237	struct dsxt	187
scanIoInit	241	struct putNotify	218
scanIoRequest	241	synchronous device support example	181
scanOnce	243	SYS_PROD_LIBS	57, 68
scanOnceSetQueueSize	120, 243	system	247
scanpel	150	<hr/> T <hr/>	
scanPeriod	238	T_A	37
scanpiol	150	Table of Makefile definitions	62
scanppl	150	TARGETS	59, 72
SCH2EDIF_FLAGS	71	task_params.h	305
SCRIPT	52	taskwd.h	232
SCRIPT_HOST	53	taskwdAnyInsert	233
SCRIPT_IOC	53	taskwdAnyRemove	233
SCRIPTS	70	taskwdInsert	232
Scripts	52	taskwdRemove	232
SCRIPTS_HOST	70	TCL libraries	60
SCRIPTS_IOC	70	TCLINDEX	60, 70
secPastEpoch	292	TCLLIBNAME	60, 70
SHARED_LIBRARIES	33, 46, 64	TEMPLATES	54, 71
show	247	Templates	54
SHRLIB_VERSION	50, 64	templates	257
Site.cshrc	76	TEMPLATES_DIR	71
Site.profile	76	TERM	248
size - field definition rules	96	Test Products	58
size_value - field definition	98	Test Scripts	59
Slope Conversion	107	testAbort	275
SNCFLAGS	71	TESTCLASSES	71
Source file dirs	40	testDiag	275
SPC_ALARMACK	97	testDone	275
SPC_AS	97	testFail	275
SPC_CALC	98	TESTJAVA	61
SPC_DBADDR	97	testOk	275
SPC_LINCONV	98	testOk1	275
		testOkV	275

testPass	275
testPlan	275
TESTPROD	58, 63
TESTPROD_HOST	58, 63
TESTPROD_IOC	58, 63
TESTSCRIPTS	59, 63
TESTSCRIPTS_HOST	59, 63
TESTSCRIPTS_IOC	59, 63
testSkip	275
testTodoBegin	275
testTodoEnd	275
top	31
Tornado II	33
tpn	156
TRAD	69
TRAPWRITE	127, 142
truncateFile	275
tsBTree.h	259
TSConfigure	121
TSconfigure	121
tsDLList.h	259
TSdriver.html	76
tsFreeList.h	259
tsMinMax.h	259
TSreport	150
tsSLList.h	259
type - variable definition	102

U

UAG	126–128
UDF	172
udf	172
uninstall	39
Unquoted String	93
USER_DBDFLAGS	42
USER_VPATH	72
USR_CFLAGS	45, 68
USR_CPPFLAGS	45, 68
USR_CXXFLAG	45
USR_CXXFLAGS	68
USR_DBDFLAGS	71
USR_INCLUDES	44, 68
USR_JAVACFLAGS	61, 72
USR_JAVAHFLAGS	62, 72
USR_LDFLAGS	45, 69
USR_LIBS	49–50, 57, 67
USR_OBJLIBS	48, 55, 66
USR_OBJS	48, 55, 65
USR_SRCS	47, 56, 65
USR_SYS_LIBS	50, 58, 67

V

VALID_BUILDS	37
var	247
variable	15, 93
variable_name - variable definition	102
veclist	153, 305
VME_IO - link field value	105

vxComLibrary	305
VXI_IO - link field value	106
vxWorks	33
vxWorks startup command file	117

W

win32.bat	77
-----------	----

Y

YACCOPT	71
---------	----